develop

The Apple Technical Journal

Issue 29 March 1997

Easy 3D With the QuickDraw 3D Viewer

Gearing Up for Asia With the Text Services Manager and TSMTE

High-Performance ACGIs in C

Using Newton Internet Enabler to Create a Web Server

Making the Most of Memory in OpenDoc

$10.00

develop

Printed on recycled paper by Stream International, USA

THINGS TO KNOW

develop, The Apple Technical Journal, a quarterly publication of the Apple Developer Relations group, is published in March, June, September, and December. It provides developers of Apple-platform products with technical articles and code that have been reviewed for robustness by Apple engineers.

All issues of *develop*, along with the code they describe, can be found on the *develop Bookmark* CD, the Reference Library edition of the *Developer CD Series*, and the Internet. The code is updated regularly, so always use the latest version.

This issue's CD. Subscription issues of *develop* are accompanied by the *develop Bookmark* CD. This CD contains a subset of the materials on the *Developer CD Series*, which is part of the Apple Developer Mailing available through the *Apple Developer Catalog*. The CD also contains Technotes, sample code, and other documentation and tools (these contents are subject to change). Items referred to as being on "this issue's CD" are located either on the Bookmark CD or on the Reference Library or Tool Chest edition of the *Developer CD Series*.

Subscriptions and back issues. You can subscribe to *develop* through the *Apple Developer Catalog* (see below, or use the subscription card in this issue). Back issues, in addition to being available electronically, can also be ordered through the catalog. The one-year U.S. subscription price is $30 (for four issues and four *develop Bookmark* CDs), or $50 U.S. in other countries. Back issues are $13 each. These prices include shipping and handling. For Canadian orders, the subscription price includes GST (R100236199).

WHERE TO FIND US

What	Who/where
develop on the Web and ftp	http://www.devworld.apple.com/develop/ ftp://ftpdev.info.apple.com/Developer_Services/Periodicals/develop/
Bookmark CD contents	http://www.devworld.apple.com/ ftp://ftpdev.info.apple.com/Developer_Services/
Technotes on the Web	http://www.devworld.apple.com/dev/technotes.shtml
Editorial comments or suggestions	Caroline Rose, crose@apple.com, (408)974-0544 fax
Technical questions about *develop*	Dave Johnson, dkj@apple.com, (408)974-0544 fax
Article submissions	develop@apple.com (ask for our Author's Kit)
Apple Developer Catalog (*develop* subscription, back issues, or other products)	http://www.devcatalog.apple.com order.adc@apple.com 1-800-282-2732 U.S., 1-800-637-0029 Canada, (716)871-6555 internationally, (716)871-6511 fax *Apple Developer Catalog*, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319
Subscription changes or queries	order.adc@apple.com *Please be sure to include your name, address, and account number as they appear on your mailing label.*

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

The nerd market has been saturated — you all have computers — but what about the "home market"? I'm no expert on this, but now that I've taken on the persona of home user myself, I feel better qualified to say why normal people would rather use calculators and play board games. I recently upgraded to a Power Mac 8500 so that I could work at home and also to be able to enjoy some fun software like the newly released electronic version of my favorite board game. Rather than just download software from an Apple server to use for work, I was going to buy something to play with to help justify my new expensive hardware purchase — just like any Mom, Pop, or kid might do.

I took a trip — several, actually — to a barn-like emporium that is Nerd Central in my neighborhood but a singularly unpleasant place for non-geeks. (The clerks seem to know that if you have to ask for what you need, you don't belong there, so they ignore you.) The first few times, I was dismayed to find that the game I wanted had been released only for Windows and not yet for Macintosh. Finally, when I looked more closely at the product in the Windows software section, I saw the small print on it that said "Windows and Mac." Then all I had to do was wait a half hour in line to buy it.

Remember the big furor over wasteful packaging when CDs started replacing record albums in audio stores? Before long the shelves were redesigned and the extra packaging eliminated. What did I find in this big box containing a software product for which I had shelled out 50-plus dollars? Nothing but a CD in its case and a huge piece of molded black plastic cleverly designed to take up all the rest of the space in the box. I wondered how much that extra packaging had cost me, but decided to get on with it and chill out over a good game.

I'll run out of space before I can get through all the other problems I encountered. Suffice it to say I was dissatisfied with the product, to the point where I would not have purchased it had I known of its shortcomings ahead of time. It paled in comparison to a similar shareware game I'd been using (for which, needless to say, I didn't have to drive to a store, search for the product, wait in line, spend a lot of money, and throw away most of the package).

The home market won't really take off until the experience of purchasing software becomes more pleasurable and foolproof. This could mean something more like buying audio CDs, where you go to a cool store organized by content type and not hardware type, where you can easily return the product within a short time for any reason, and where you can even try out the disc before buying it. Or it may mean waiting till the practice of buying software off the Web has taken hold — especially inexpensive shareware, so that satisfaction is guaranteed in advance. Only solutions like these will roust the home market. Otherwise it's just too much of a pain.

Crose

Caroline Rose
Editor

CAROLINE ROSE (crose@apple.com) isn't a typical home user, since she was one of the first people to use a Macintosh and she cut her teeth before that on a timesharing system. But that was all in the line of duty, whereas at leisure she uses computers as little as possible. She's even been known to handwrite letters to friends. Speaking of which, Caroline has received so little mail from *develop* readers lately that this issue is missing a Letters section and she's missing hearing from you. What's on your mind about *develop*? Please take a moment to let Caroline know. •

Looking to complete the set?

develop

VR movies
ral diverse
with extensive
3D data sets
QuickTime VR
movies from their data sets
to show potential customers;
the movies display the modeled
objects more effectively 2D
representation and compromise
the data in the process.
Archaeologists can QuickTime
VR movies to record site information
during digs, realtors can use them to
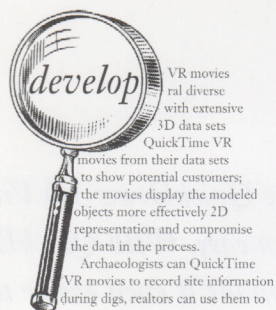
If you're looking for a complete *develop* collection, full-color, bound copies are available for $13 per issue, including shipping and handling. (Back issues are also on the *develop* Bookmark CD and the *Developer CD Series* Reference Library edition, as well as on the Internet.) For more information about how to order printed back issues (and where to find them online), see the inside front cover of this issue. *Supplies are limited. Please allow 4 to 6 weeks for delivery.*

Easy 3D With the QuickDraw 3D Viewer

Ever since QuickDraw 3D shipped in 1995, the QuickDraw 3D Viewer has made adding 3D support to your application easy. With QuickDraw 3D version 1.5 we've enhanced the Viewer to make it even easier to use. We've improved the user interface, added support for Undo, and rolled in some new API calls. Here you'll learn how to implement the Viewer to provide simple yet powerful 3D capabilities in your products.

NICK THOMPSON

The QuickDraw 3D Viewer provides a way for you to add 3D support to your application without having to come to grips with the complexity of the whole QuickDraw 3D programming API. As described in "QuickDraw 3D: A New Dimension for Macintosh Graphics" in *develop* Issue 22, full use of QuickDraw 3D requires you to understand many things before you can get started; for example, you need to be able to set up data structures to hold not only the geometries being modeled but also the other elements of a scene, including the lighting, the camera, and the draw context. But sometimes you just want to be able to display some 3D data in your application without having to write five pages of setup code.

If this situation sounds familiar to you, the Viewer is tailor-made for your application. You'll learn all you need to know to be able to use it from reading this article and examining the accompanying sample applications. Still, you might want to read the article in Issue 22 as background and to get a sense of how you can use the Viewer in conjunction with the QuickDraw 3D shared library.

ABOUT THE VIEWER

The QuickDraw 3D Viewer is a high-level shared library, available in both Macintosh and Windows versions, that's separate from the QuickDraw 3D shared library. With fewer calls than the full QuickDraw 3D API, the Viewer is a great place to start exploring QuickDraw 3D. By implementing the Viewer, you can enable users to view and have a basic level of interaction with 3D data in your application without having to call any QuickDraw 3D functions. When you need more power, you can always mix QuickDraw 3D calls with Viewer calls.

The Viewer is ideal for applications that might be described as traditional 2D applications, such as image database and page layout applications. For example, the

NICK THOMPSON (nickt@apple.com) went last summer to New Orleans, a city with a great public aquarium, with the rest of the QuickDraw 3D team. He spent a lot of time looking at totally awesome products from other vendors and drooling over the SGI Onyx Infinite Reality demo.

He also spent time at the aquarium, feeding his fascination with the ocean and its life forms, and brought two fish tanks back with him — one for his home and another for his office. This way, if he can't be in the surf, he at least has props for his fantasies about being there. •

image database Cumulus (from the German developer Canto Software GMbH) is a traditional 2D application that implements the Viewer to enable users to manipulate objects in 3D (see Figure 1).
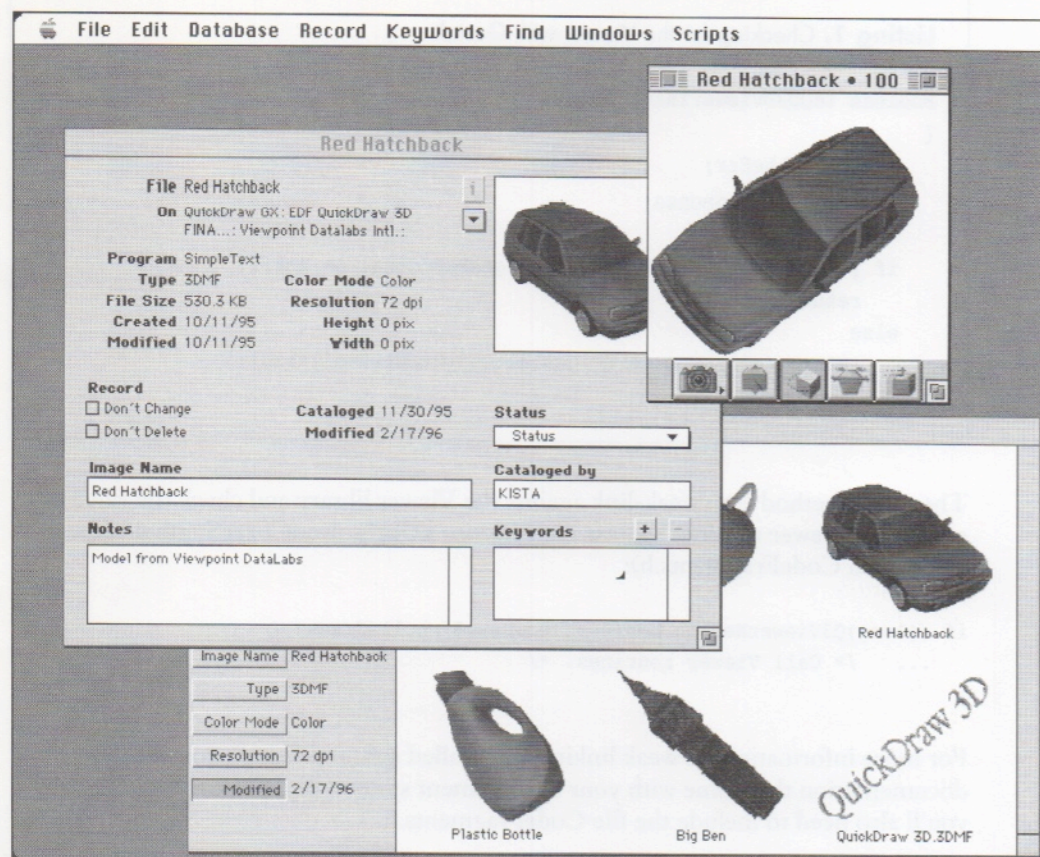
Figure 1. An example of Viewer use in the Cumulus image database

The Viewer gives your application considerable functionality for free. For example, the Macintosh version of the Viewer supports drag and drop of 3D data. And the Viewer allows access to the view object (described in detail in the article in *develop* Issue 22) so that you can add to your application the capability of changing the lighting, the camera angles and position, and other things such as the type of renderer being used.

Implementing the Viewer in your application is simple. After going over a few preliminaries, we'll look in detail at two sample applications — one just a bare-bones framework for using the Viewer, and the second a more elaborate application that implements a fuller set of Viewer features. The source code for both programs accompanies this article on this issue's CD and *develop*'s Web site.

CHECKING THAT THE VIEWER IS INSTALLED

Before you can use the Viewer, you need to make sure that it's installed. There are two ways to do this on the Macintosh: you can use Gestalt on System 7 or you can weak-link against the library and check to see if one of the Viewer routines has been declared when you launch your application.

You need to call Gestalt with the constant gestaltQD3DViewer, as shown in Listing 1. The routine IsQD3DViewerInstalled returns a Boolean indicating whether the Viewer

has been installed correctly. The bit selector gestaltQD3DViewerAvailable can be used to test the appropriate bit of the response from Gestalt.

```
Listing 1. Checking for the Viewer with Gestalt

Boolean IsQD3DViewerInstalled()
{
    OSErr   theErr;
    long    gesResponse;

    if (Gestalt(gestaltQD3DViewer, &gesResponse) != noErr)
        return false;
    else
        return (gesResponse == gestaltQD3DViewerAvailable);
}
```

The other method is to weak-link against the Viewer library and check the value of one of the Viewer routines against the constant kUnresolvedCFragSymbolAddress (defined in CodeFragments.h):

```
if ((long)Q3ViewerNew != kUnresolvedCFragSymbolAddress) {
    ...   /* Call Viewer routines. */
}
```

For more information on weak linking (also called soft importing), consult the documentation that came with your development system. If you use this method, you'll also need to include the file CodeFragments.h.

DETERMINING THE VIEWER VERSION

Version 1.5 of the Viewer introduces several new API features not found in previous versions of the Viewer. If you want your application to be compatible with previous versions of the Viewer, you need to check the version by calling the new routine Q3ViewerGetVersion. Of course, before you can call this routine, you'll need to test whether it's been loaded along with the Viewer shared library by checking its address against the symbol kUnresolvedCFragSymbolAddress. If it hasn't been loaded, you can safely assume that the Viewer version is 1.0.

Alternatively, you can check the address of each function you need to use against kUnresolvedCFragSymbolAddress. Listing 2 shows a routine to determine the Viewer version; this routine works with all versions of the Viewer library.

A BARE-BONES FRAMEWORK FOR USING THE VIEWER

Now let's take a look at one of the simplest possible applications we might write to enable someone to open and view QuickDraw 3D metafiles (files containing 3DMF data). Of course, this isn't a real Macintosh program — it opens only one document, it doesn't respond to Apple events, it doesn't present a menu bar, and the user can't save changes made in the window. But it does demonstrate that with just five calls to the Viewer library you can provide good support for 3DMF data in your application. We're not going to cover anything but the QuickDraw 3D part of this application in any detail, but the source code is commented well enough so that it should be clear how it works.

Listing 2. Checking the Viewer version number

```
OSErr GetViewerVersion(unsigned long *major, unsigned long *minor)
{
    /* Version 1.0 of the QuickDraw 3D Viewer had no get version call, so
       see if the symbol for the API routine descriptor is loaded. */
    if ((Boolean)Q3ViewerGetVersion == kUnresolvedCFragSymbolAddress) {
        *major = 1;
        *minor = 0;
        return noErr;
    }
    else
        return Q3ViewerGetVersion(major, minor);
}
```

THE WINDOW

Figure 2 shows the window from our simple application, called BareBones3DApp. An instance of the Viewer — a viewer object — can occupy an entire window or it can occupy some smaller portion of a window. In the case of BareBones3DApp, the viewer object entirely fills the window. The viewer object consists of a controller strip and a content area outlined with a drag border.
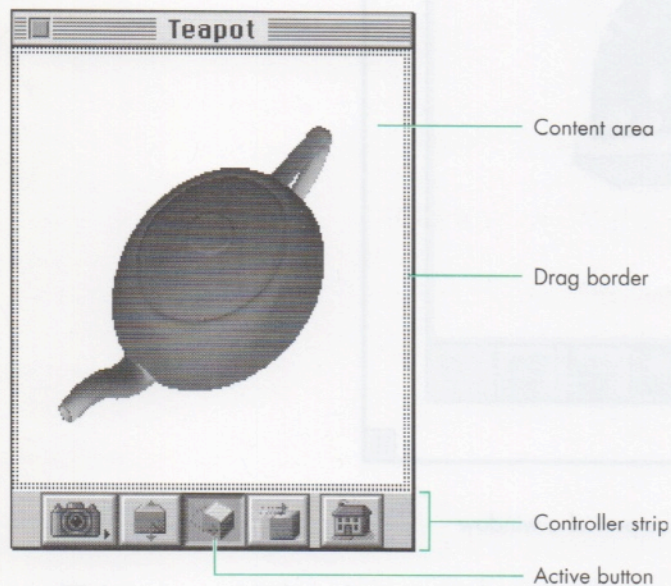
Figure 2. Window from BareBones3DApp

- The *controller strip* contains a number of buttons for manipulating the user's point of view (that is, the view's camera). Each of the buttons either performs a specific function, such as setting a particular camera for the view, or sets a mode that determines how user interactions are handled. The controller strip can also be hidden; in this case, a visual element known as a *badge* takes its place to indicate to the user that the image in the window represents a 3D model. The user can click on the badge to make the controller strip appear.

- The *content area* (called the *picture area* in earlier documentation) is where the 3DMF data is drawn. Users can interact with the object drawn in the content

area in one of several modes, the modes being selected by clicking one of the buttons in the controller strip. In the default mode that the window opens up in, users can change the camera angle by dragging across the object.

- The "OpenDoc-style" *drag border* indicates that the viewer content area can initiate drags of 3DMF data. By dragging on this border the user can drag the object displayed in the content area. If dragging into and out of the content area is enabled (as it is by default), the border will be highlighted when a drag is initiated to indicate that the content area can receive drops as well.

The part of the window that contains the content area and the controller strip (if present) is the *viewer pane*. As an alternative to having the viewer pane entirely fill the window, you can place the viewer pane in just part of the window, as shown in Figure 3. This is useful for embedding a 3D picture in a document window.
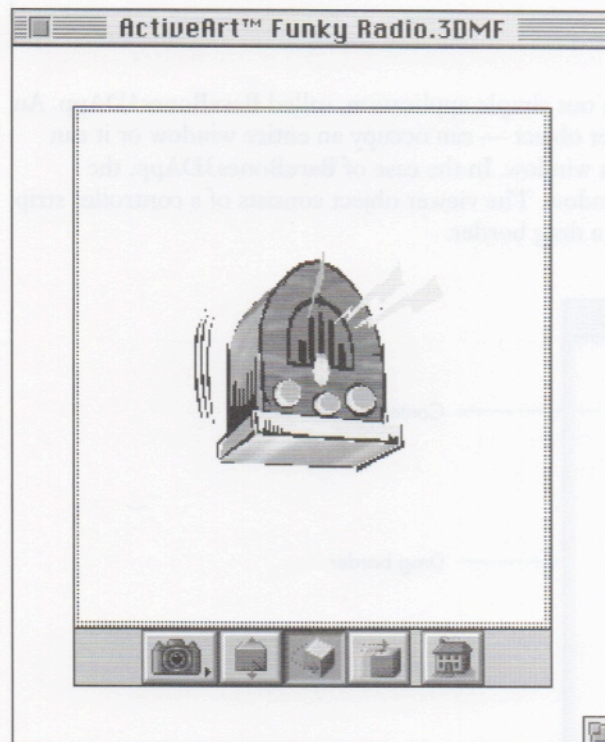
Figure 3. The viewer pane as part of a window

In the controller strip, the active button is drawn to look as if it's been pressed. The buttons shown in Figures 2 and 3 are the default ones; you can hide those you don't want, or make visible the one additional button that's hidden by default, by setting flags that will be discussed shortly. You can also hide or show the entire controller strip; you'll see how to do this later.

The full set of available controller buttons is shown in Figure 4. Let's look at each in turn.

The *camera viewpoint button* (called the *camera angle button* in earlier documentation) enables the user to view the displayed object from a different camera angle. Holding down the button causes a pop-up menu to appear, listing the predefined direction cameras as well as any
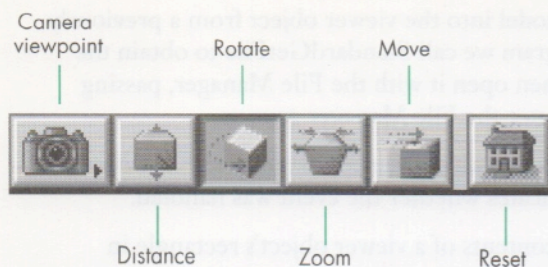
Figure 4. The full set of available controller buttons

perspective (view angle aspect) cameras stored in the view hints of the 3DMF data. If any such cameras have name attributes associated with them in the data, the names are displayed in the pop-up menu; otherwise, the cameras are listed as "Camera #1," and so on. (The predefined direction cameras are calculated based on the front and top custom attributes if present in the 3DMF view hints; otherwise, they're calculated from the displayed object's coordinate space.)

The *distance button* lets the user move the displayed object closer or farther away. Clicking the distance button and then dragging downward in the content area moves the object closer. Dragging upward in the content area moves the object farther away. The Down Arrow and Up Arrow keys also move the object closer or farther away, respectively.

The *rotate button* enables rotating an object. Clicking this button and then dragging in the content area rotates the displayed object in the direction of the drag. The arrow keys rotate the object in the direction of the arrow. With version 1.5 of the Viewer library, you can use the Shift key to constrain the motion of the object as you rotate it.

The *zoom button* enables the user to alter the field of view of the current camera, thereby zooming in or out on the displayed object. After the zoom button is clicked, pressing the Up Arrow and Down Arrow keys zooms the object out and in. By default, this button isn't displayed.

The *move button* lets the user move an object. Clicking this button and then dragging in the content area moves the object to a new location. The arrow keys move the object in the direction of the arrow.

The *reset button* resets the camera angle and position to their initial settings.

THE BASIC CALLS

As mentioned earlier, you can add support for 3DMF data with calls to just five routines in the Viewer shared library. These routines, described below, are the ones we use in BareBones3DApp. For more details on these calls, see the book *3D Graphics Programming With QuickDraw 3D*.

- Q3ViewerNew — Creates a viewer object and attaches it to a previously created window, then returns a reference to the viewer object. You need to pass this reference to other Viewer routines.

- Q3ViewerDispose — Disposes of the viewer object and associated storage. You'll probably want to do this just before closing and disposing of the window.

- Q3ViewerSetFile — Loads a model into the viewer object from a previously opened 3DMF file. In our program we call StandardGetFile to obtain the details of the file to open and then open it with the File Manager, passing Q3ViewerSetFile the file reference the File Manager gave us.

- Q3ViewerEvent — Gives the viewer object the opportunity to handle events, then returns a Boolean that indicates whether the event was handled.

- Q3ViewerDraw — Draws the contents of a viewer object's rectangle in response to an update event.

THE MAIN ROUTINE

The main routine of BareBones3DApp handles initialization of Macintosh managers, grows the heap to its maximum size, and checks to see if the QuickDraw 3D Viewer is installed. There must be at least 24K free in the application heap before a call to Q3ViewerNew can succeed, so it's important to call the Toolbox routine MaxApplZone to grow the application heap to its maximum size at the start of the program. Otherwise, the Viewer may detect (in error) that there's not enough memory to run.

The program then calls the Toolbox routine StandardGetFile to locate a 3DMF file to open and read. The selected file is opened, and a window is created. The routine to create a viewer object looks like this:

```
TQ3ViewerObject Q3ViewerNew(CGrafPtr port, Rect *rect, unsigned long flags);
```

Notice that you need to pass in port, rectangle, and flags parameters. It's possible to create an "empty" viewer object by passing in nil for the port parameter; you can then assign a port later with Q3ViewerSetPort. The flags parameter is used to set flags that control various aspects of the behavior of the viewer object you create; these flags, along with the behavior that results when they're set, are listed in Table 1. The flags of an already created viewer object can be changed with the Q3ViewerSetFlags routine.

Table 1. Flags that control aspects of the viewer object

Flag	Result when set	Default
kQ3ViewerActive	The viewer object is active (can be manipulated).	On
kQ3ViewerShowBadge	A badge is displayed in the viewer pane. This flag should be cleared when kQ3ViewerControllerVisible is set.	Off
kQ3ViewerControllerVisible	The controller strip is visible. This flag should be cleared when kQ3ViewerShowBadge is set.	On
kQ3ViewerDrawFrame	A one-pixel frame is drawn within the viewer pane.	Off
kQ3ViewerDraggingOff	Dragging into and out of the viewer content area is disabled.	Off
kQ3ViewerDraggingInOff	Dragging into the viewer content area is disabled.	Off
kQ3ViewerDraggingOutOff	Dragging out of the viewer content area is disabled.	Off
kQ3ViewerButtonCamera	The camera viewpoint button in the controller strip is visible.	On
kQ3ViewerButtonTruck	The distance button in the controller strip is visible.	On
kQ3ViewerButtonOrbit	The rotate button in the controller strip is visible.	On
kQ3ViewerButtonZoom	The zoom button in the controller strip is visible.	Off
kQ3ViewerButtonDolly	The move button in the controller strip is visible.	On
kQ3ViewerButtonReset	The reset button in the controller strip is visible.	On
kQ3ViewerOutputTextMode	Q3ViewerWriteFile and Q3ViewerWriteData write out 3DMF data in text mode.	Off

Table 1. Flags that control aspects of the viewer object *(continued)*

Flag	Result when set	Default
kQ3ViewerDragMode	The viewer object responds only to drag and drop interaction, and can't be manipulated in any other way. A mouse-down in the content area will initiate a drag operation.	Off
kQ3ViewerDrawGrowBox	The viewer object displays a size box in the lower-right corner.	Off
kQ3ViewerDrawDragBorder	The viewer object displays a drag border around the perimeter of the content area.	On
kQ3ViewerDefault	Returns the viewer object to the default configuration.	

The flags kQ3ViewerButtonTruck, kQ3ViewerButtonOrbit, kQ3ViewerButtonZoom, and kQ3ViewerButtonDolly can also be used with the Q3ViewerSetCurrentButton routine. Passing one of these flags to this routine sets the viewer object to the mode indicated by the button. If the button is visible in the controller strip, it's drawn to look as if it's been pressed, and the previously selected button is deselected.

You can override the default drag-handling behavior by attaching your own drag handler to the document window. You'll want to do this if your application supports multiple viewer objects per window or if you're creating something where the default may get in the way of your programming model — for example, an OpenDoc part or a HyperCard XCMD.

Listing 3 shows how we implement the main routine in C. Note that we place a reference to the viewer object in the window's refCon field so that later in the program we can easily get the viewer object associated with the window.

Listing 3. The main routine from BareBones3DApp

```
void main(void)
{
    short              myNumTypes = 1, myRefNum;
    SFTypeList         myTypeList = { '3DMF' };
    StandardFileReply  mySFReply;
    OSErr              theErr = noErr;
    WindowPtr          myWind = nil;
    Rect               myRect = { 0, 0, kWindHeight, kWindWidth };
    TQ3ViewerObject    myViewer;

    /* Initialize all the needed managers. */
    InitGraf((Ptr)&qd.thePort); InitFonts(); InitWindows();
    InitMenus(); TEInit(); InitDialogs((long)nil);
    InitCursor();

    /* Expand the heap to maximum size. */
    MaxApplZone();

    /* We weak-linked against the Viewer. Now check that it's installed. */
    if ((long)Q3ViewerNew != kUnresolvedCFragSymbolAddress) {
        StandardGetFile(nil, myNumTypes, myTypeList, &mySFReply);
```

(continued on next page)

```
        if (mySFReply.sfGood) {
            theErr = FSpOpenDF(&mySFReply.sfFile, fsRdPerm, &myRefNum);
            OffsetRect(&myRect, 50, 50);
            myWind = NewCWindow(nil, &myRect, "\pViewerApp", true,
                            documentProc, (WindowPtr)-1, true, 0L);
            if (myViewer = Q3ViewerNew((CGrafPtr)myWind, &myWind->portRect,
                        kQ3ViewerDefault)) {
                /* If the viewer object isn't nil, we created it OK. */
                theErr = Q3ViewerUseFile(myViewer, myRefNum);
                SetWRefCon(myWind, (long)myViewer);
                MainEventLoop();
            }
        }
    }
    ExitToShell();
}
```

THE MAIN EVENT LOOP

The main event loop, shown in Listing 4, handles events until the window is closed. There are only two types of event that we'll consider handling in this program: update and mouse-down events. In response to an update event we'll need to call Q3ViewerDraw. Handling mouse-down events is somewhat more complex, since we'll need to determine where the mouse-down occurred.

```
void MainEventLoop(void)
{
    WindowPtr         myWind;
    Boolean           gotEvent;
    TQ3ViewerObject   theViewer;
    OSErr             theErr;
    RgnHandle         tempRgn;
    Rect              dragRect;
    EventRecord       theEvent;
    GrafPtr           savedPort;

    while ((myWind = FrontWindow()) != nil) {
        gotEvent = WaitNextEvent(everyEvent, &theEvent, GetCaretTime(),
            nil);
        if (gotEvent) {
            switch (theEvent.what) {
                case updateEvt:
                    myWind = (WindowPtr)theEvent.message;
                    theViewer = (TQ3ViewerObject)GetWRefCon(myWind);
                    BeginUpdate(myWind);
                    theErr = Q3ViewerDraw(theViewer);
                    EndUpdate(myWind);
                    break;
```

(continued on next page)

Listing 4. The main event loop from BareBones3DApp *(continued)*

```
            case mouseDown:
                switch (FindWindow(theEvent.where, &myWind)) {
                    case inGoAway:
                        theViewer = (TQ3ViewerObject)GetWRefCon(myWind);
                        theErr = Q3ViewerDispose(theViewer);
                        DisposeWindow(myWind);
                        break;
                    case inContent:
                        GetPort(&savedPort);
                        SetPort((GrafPtr)myWind);
                        Q3ViewerEvent(theViewer, &theEvent);
                        SetPort(savedPort);
                        break;
                    case inDrag:
                        tempRgn = GetGrayRgn();
                        dragRect = (**tempRgn).rgnBBox;
                        DragWindow(myWind, theEvent.where, &dragRect);
                        break;
                }
                break;
            }
        }
        SetPort(savedPort);
    }
}
```

- If the mouse-down was in the close box of the window, we need to dispose of the viewer object and the window. In the main event loop, we check to see if there's a window open for the application by calling the Toolbox routine FrontWindow; if there isn't one open, the application quits.

- If the mouse-down was in the content area of the window, we can pass the event record to the routine Q3ViewerEvent to handle. For version 1.0.4 and earlier versions of QuickDraw 3D, you also need to ensure that the port is set to the current window, as shown in Listing 4, for Q3ViewerEvent to work as expected.

- If the mouse-down was in the title bar of the window, we need to drag the window around until the user releases the mouse button. Fortunately, there's a Toolbox routine to do this — DragWindow. Notice that we pass in the rectangle associated with the desktop region; this works well for the case where multiple monitors are attached to the computer.

A FULL-FEATURED APPLICATION USING THE VIEWER

Our second sample application, called FullFeatured3DApp, goes much of the way toward providing the kind of features that you'd expect in a real application. It also gives some examples of how to use the full QuickDraw 3D library in conjunction with the Viewer library. Multiple 3DMF documents can be opened and changes can be saved; Undo, Cut, Copy, and Paste are supported; the user can change the viewer background color and the renderer type; and you can show and hide the buttons in the controller strip and even the strip itself. I'm not going to show all of the code here, but I'll cover the salient points of the application, starting with the basics and

A LOOK AT THE QUICKDRAW 3D VIEWER FOR WINDOWS
BY JOHN LOUCH

The QuickDraw 3D Viewer for Windows differs from its Macintosh cousin in a number of ways. In fact, from an API and functional standpoint, the Windows Viewer differs from the Macintosh Viewer more than the Windows version of any other QuickDraw 3D component — including the QuickDraw 3D core library, QuickDraw 3D RAVE, the interactive renderer, and the 3D Viewer Controller — differs from the Macintosh version. We'll look at these differences here.

Most fundamentally, all routines are renamed in the Windows Viewer to begin with "Q3WinViewer" instead of "Q3Viewer," to prevent name-space collisions. The Windows Viewer is actually implemented as a Windows control window (similar to the common controls, like the hierarchical tree view, that were included with Windows 95). The Windows Viewer can be implemented with the QuickDraw 3D Viewer API or the standard Windows API.

If the QuickDraw 3D Viewer is a Windows pop-up window, it can be implemented using these few calls:

- Q3WinViewerNew (or the Win32 call CreateWindow, passing in the constant kQ3ViewerClassName) — Creates a viewer object.

- WM_SYSCOLORCHANGE and WM_SETFOCUS — The parent window must post these messages to the viewer window (using PostMessage or SendMessage) when it receives them.

Because the Windows Viewer is a window class, you don't need to send it events or ask it to update or draw. Those functions are all handled automatically by the Windows windowing system. Of course, you can still call Q3WinViewerMouseDown/MouseUp/ContinueTracking at any time.

The following flags used by the Macintosh Viewer don't apply to the Windows Viewer: kQ3ViewerDraggingOff, kQ3ViewerDragMode, kQ3ViewerDrawGrowBox, kQ3ViewerDrawDragBorder, kQ3ViewerDraggingOutOff. Most of these flags relate to drag and drop; the Windows Viewer doesn't support dragging out of the viewer content area as the Macintosh Viewer does. The other flags relate to human interface differences between the two systems.

The following Windows Viewer functions differ in some way from their Macintosh counterparts:

- Q3WinViewerNew(HWND window, const Rect *rect, unsigned long flags) — Takes an HWND instead of a

CGrafPtr. If the window parameter is NULL, a parentless pop-up window is created; otherwise, the viewer window created is owned by the HWND you pass in and is a child window. The flags parameter is also a little different in Windows. You can add in any of the standard Windows window-style flags (such as WS_CHILD) with bitwise-OR to affect the type of window that you get.

- Q3WinViewerUseFile and Q3WinViewerWriteFile — Identical to their Macintosh counterparts except they require a Windows file handle; for example, Q3WinViewerWriteFile(TQ3ViewerObject viewer, HANDLE fileHandle).

- Q3WinViewerSetFlags — The parameters for this function are the same as for the Macintosh version. The behavior is different when you set the flags that show or hide controller buttons (kQ3ViewerButtonCamera, kQ3ViewerButtonTruck, and so on). On the Macintosh, you must force a redraw (with Q3ViewerDraw or Q3ViewerDrawControlStrip) after you change which buttons are shown. In Windows the redraw happens automatically inside this call.

- Q3WinViewerGetMinimumDimensions — The behavior of this function is different from that of the Macintosh function because the window has to be shown with a toolbar to calculate the minimum dimensions.

- Q3WinViewerGetWindow — Returns the HWND of the viewer window and not the parent window.

The following functions are new in the API for the Windows Viewer:

- Q3WinViewerGetControlStrip — Returns the HWND of the controller strip, which is an actual Windows toolbar common control. With this function you can get an HWND reference and then actuate on it with the Windows API.

- Q3WinViewerGetBitmap — Returns a 32-bit-deep bitmap of the current model associated with the viewer object.

- Q3WinViewerGetViewer(HWND theWindow) — Returns the TQ3ViewerObject that's associated with a window, if that window is a Viewer Window class.

- Q3WinViewerSetWindow — Sets the window in which the viewer will draw. This function is almost identical to Q3ViewerSetPort except for the semantic differences between platforms.

then showing how to implement the various Viewer features. Again, the code accompanying this article is well commented so you should have no problem following what's going on.

I'm not going to show you a sample application that uses the Windows Viewer, but you can get a good idea of how it differs from the Macintosh Viewer by reading "A Look at the QuickDraw 3D Viewer for Windows."

THE BASICS
The first thing we do is to define a simple structure to store the information we need for each 3DMF document. In a more substantial application you could add fields here as required. We'll need to store a reference to the viewer object and also some information about the file the model came from, so that we can implement the Save and Revert commands. The definition for this structure is as follows:

```
typedef struct {
  TQ3ViewerObject    fViewer;     /* reference to the viewer object */
  FSSpec             fFSSpec;     /* reference to the file for the document */
} ViewerDocument, *ViewerDocumentPtr, **ViewerDocumentHdl;
```

We're creating three new types here: a document record plus a pointer and a handle to that document record. In the sample code for this article we generally put the document-related information in a Macintosh handle and store this handle in the refCon field of that document's window. That way we can easily get at the information we need. As shown in Listing 5, creating a window then becomes a matter of creating the handle for the document record with NewHandleClear (which zeros out the allocated handle), creating a window for the document with NewCWindow, creating a viewer object with Q3ViewerNew and associating the window with the viewer object, and finally storing the handle to the document in the window's refCon field with the handy utility function SetWRefCon.

SetWRefCon has a sister function called GetWRefCon, and we'll use this whenever we need to get the viewer object associated with a window. Once we have a WindowPtr reference to a window, getting the associated viewer object is a question of getting the value from the window's refCon field, casting it to a ViewerDocumentHdl, and getting the viewer object from the appropriate field.

```
theViewerDocumentHdl = (ViewerDocumentHdl)GetWRefCon(theWindow);
if (theViewerDocumentHdl != NULL) {
   if ((theViewer = (**theViewerDocumentHdl).fViewer) != NULL) {
      ...   /* Your code to work with the viewer object */
   }
}
```

The next few sections look at how we use functions from the QuickDraw 3D Viewer shared library to add cool features to our program.

READING AND WRITING 3DMF FILES
Reading files with the Q3ViewerUseFile routine is one way of getting 3DMF data into your viewer object, as we saw in Listing 3. There are other I/O routines we can use for writing to a file, and for reading from and writing to areas of memory.

- Q3ViewerUseData — Similar to Q3ViewerUseFile, except that instead of a file reference it takes a pointer to 3DMF data stored in memory and displays that data in a viewer object you create.

Listing 5. Creating a window

```
WindowPtr DoCreateNewViewerWindow(unsigned char *windowName)
{
    WindowPtr         theWindow;
    Rect              myRect = { 0, 0, kWindHeight, kWindWidth };
    TQ3ViewerObject   myViewer;
    ViewerDocumentHdl myViewerDocument = NULL;

    /* Create a document record to hold the data for this instance. */
    myViewerDocument =
        (ViewerDocumentHdl)NewHandleClear(sizeof(ViewerDocument));

    /* Ideally, we should stagger the rect. */
    OffsetRect(&myRect, 50, 50);

    theWindow = NewCWindow(NULL, &myRect, windowName, true,
        documentProc, (WindowPtr)-1, true, 0L);

    /* Create the viewer object associated with this window. */
    if ((myViewer = Q3ViewerNew((CGrafPtr)theWindow,
            &theWindow->portRect, kQ3ViewerDefault)) != NULL) {
        /* Store a reference to the viewer object in the document
           structure. */
        (**myViewerDocument).fViewer = myViewer;

        /* Store a reference to the document structure in the refCon field
           of the window. */
        SetWRefCon(theWindow, (long)myViewerDocument);
    }
    else {
        /* Clean up any allocated storage and quit. */
        if (myViewerDocument)
            DisposeHandle((Handle)myViewerDocument);
        if (theWindow != NULL)
            CloseWindow(theWindow);
        theWindow = NULL;
    }
    return theWindow;
}
```

- Q3ViewerWriteFile — Writes the data being displayed in a viewer object out to a file, including information about the view. We'll use this routine to implement our Save and Save As commands.

- Q3ViewerWriteData — Similar to Q3ViewerWriteFile, except that the data is written to an area of memory rather than a file.

We store a reference to a file associated with the viewer document in an FSSpec record in our document structure. This makes it a lot easier to deal with files. When we want to save a viewer document we can look at the FSSpec to get the file in which to save the document. If the FSSpec is blank, we know that the document has no file associated with it. When reading a file, we need to make sure that we store the FSSpec in our document structure, as Listing 6 illustrates.

Listing 6. Reading 3DMF data from a file

```
WindowPtr HandleFileOpenItem(FSSpec *theFSSpec)
{
    OSErr              theError;
    short              theRef;
    WindowPtr          theWindow;
    TQ3ViewerObject    theViewer;
    ViewerDocumentHdl theViewerDocumentHdl;

    /* Open the file. */
    theError = FSpOpenDF(theFSSpec, fsRdPerm, &theRef);
    if (theError == noErr) {
        theWindow = DoCreateNewViewerWindow(theFSSpec->name);
        if (theWindow != NULL) {
            theViewerDocumentHdl =
                (ViewerDocumentHdl)GetWRefCon(theWindow);
            if (theViewerDocumentHdl != NULL) {
                if ((theViewer = (**theViewerDocumentHdl).fViewer)
                        != NULL) {
                    (**theViewerDocumentHdl).fFSSpec = *theFSSpec;
                    theError = Q3ViewerUseFile(theViewer, theRef);
                    /* Ignore error. */
                }
            }
        }
        theError = FSClose(theRef);
        /* Ignore error. */
    }
    return theWindow;
}
```

In this example we open the data fork of the file selected by the user (or passed in as part of an Apple event) with FSpOpenDF and create a window with the routine DoCreateNewViewerWindow, described earlier. We then store the reference to the file in the appropriate field of the document record and read in the 3DMF data with the routine Q3ViewerUseFile.

Writing out 3DMF data is equally straightforward, as shown in Listing 7. We use the routine Q3ViewerWriteData to write the 3DMF data to a previously opened file. We use the FSSpec previously stashed in the document record to open the file, with the routine FSpOpenDF. Naturally, the Save As and Revert commands can be handled in a similar way, allowing you to implement a standard File menu with all the commands usually found there.

SUPPORTING THE CLIPBOARD

The Clipboard enables users to copy data between windows in an application and between applications that support the same data format. For example, we might want to copy data between our sample application and the standard Macintosh Scrapbook. We can do this by supporting Cut, Copy, and Paste in our application. This is really easy to do with the Viewer, which supplies a number of utility routines specifically for dealing with the Clipboard.

Listing 7. Writing 3DMF data to a file

```
OSErr HandleFileSaveItem(WindowPtr theWindow)
{
    OSErr               theError = paramErr;
    short               theRef;
    TQ3ViewerObject     theViewer;
    StandardFileReply   theSFReply;
    ViewerDocumentHdl   theViewerDocumentHdl;
    FSSpec              theFSSpec;

    /* This option can't be selected unless there's a front window.
       The option is dimmed in the routine AdjustMenus if there's no
       window. */
    if (theWindow != NULL) {  /* sanity check */
        theViewerDocumentHdl = (ViewerDocumentHdl)GetWRefCon(theWindow);
        if (theViewerDocumentHdl != NULL) {
            theFSSpec = (**theViewerDocumentHdl).fFSSpec;
            /* Open the file. */
            theError = FSpOpenDF(&theFSSpec, fsWrPerm, &theRef);
            if (theError == noErr) {
                if ((theViewer = (**theViewerDocumentHdl).fViewer)
                        != NULL) {
                    theError = Q3ViewerWriteFile(theViewer, (long)theRef);
                }
                theError = FSClose(theRef);
            }
        }
    }
    return theError;
}
```

- Q3ViewerCopy — Copies the contents of the viewer object to the desk scrap in both 3DMF and PICT formats (the latter for applications that don't support 3D data).

- Q3ViewerCut — Does the same thing as Q3ViewerCopy, but the content area of the viewer window is cleared.

- Q3ViewerPaste — If the Clipboard contains 3DMF data, replaces the data in the viewer object.

- Q3ViewerClear — Clears the content area of the viewer window and resets the default camera angle and position. This is effectively the same as "delete all" for the contents of the viewer object.

In addition, the Q3ViewerUndo routine can help you support Undo for several Viewer operations.

If your application has a standard Edit menu, handling events in this menu is simple given the routines described above. Listing 8 demonstrates how to use these routines.

Obviously, for this to work correctly the Edit menu needs to be set up so that items are dimmed and shown appropriately — Copy makes no sense for an empty viewer object, and Paste makes no sense if there's no 3DMF data to paste. So we need to do

Listing 8. Using the Clipboard utility routines

```
void HandleEditMenu(short menuItem)
{
    ViewerDocumentHdl  theViewerDocumentHdl;
    WindowPtr          theWindow;
    TQ3ViewerObject    theViewer;
    OSErr              theError;

    theWindow = FrontWindow();
    if (theWindow != NULL) {
        /* Get the reference to our viewer document data structure
           from the reference constant for the window. Cast it to the
           appropriate type. If we can't get it (if it's NULL), bail. */
        theViewerDocumentHdl = (ViewerDocumentHdl)GetWRefCon(theWindow);
        if (theViewerDocumentHdl == NULL)
            return;
        /* Get the reference to our viewer object from our data
           structure. */
        theViewer = (**theViewerDocumentHdl).fViewer;
        if (theViewer == NULL)
            return;
        switch (menuItem) {
            case iEditUndoItem:
                theError = Q3ViewerUndo(theViewer);
                Q3ViewerDrawContent(theViewer);
                break;
            case iEditCutItem:
                theError = Q3ViewerCut(theViewer);
                break;
            case iEditCopyItem:
                theError = Q3ViewerCopy(theViewer);
                break;
            case iEditPasteItem:
                theError = Q3ViewerPaste(theViewer);
                break;
            case iEditClearItem:
                theError = Q3ViewerClear(theViewer);
                break;
        }
    }
}
```

two things: check that there's some content in the viewer object, and check that there's something on the scrap that can be pasted. We do this with the routines Q3ViewerGetState and GetScrap. We then enable or disable Cut, Copy, Clear, and Paste accordingly, as illustrated in Listing 9. This listing also shows how to set up the Undo menu item.

SETTING THE VIEWER BACKGROUND COLOR

You might want to let the user set the background color of the viewer — for example, to match the background color used for a multimedia presentation or to match the color of a Web page. We use the routine Q3ViewerSetBackgroundColor to do this,

Listing 9. Setting up the Edit menu

```
/* Get the viewer state. We need to know if it's empty. */
theViewerState = Q3ViewerGetState(theViewer);
...


/* Adjust the Edit menu. */
theMenu = GetMHandle(mEditMenu);
if (((theViewerState & kQ3ViewerHasUndo) {
   /* Undo is possible; get the string for this item and enable it. */
   Boolean  canUndo;

   /* Hokeyness alert: We pass in the address of the second element of
      the itemString array, allowing us to set the length later in the
      first element of the array, saving us the need to do an in-place
      C-to-Pascal string conversion (the Toolbox routines require a
      Pascal-format string that has the same length as the first byte). */
   canUndo = Q3ViewerGetUndoString(theViewer, &itemString[1],
      &itemStringLength);
   itemString[0] = (char)itemStringLength;

   /* If we can undo, enable the new string; if not, use the default
      can't-undo string. */
   if (canUndo == true && itemStringLength > 0) {
      SetMenuItemText(theMenu, iEditUndoItem,
         (unsigned char *)itemString);
      EnableItem(theMenu, iEditUndoItem);
   }
   else {
      GetIndString((unsigned char *)itemString, 2223, 1);
      SetMenuItemText(theMenu, iEditUndoItem,
         (unsigned char *)itemString);
      DisableItem(theMenu, iEditUndoItem);
   }
}
else {   /* Undo isn't possible. */
   GetIndString((unsigned char *)itemString, 2223, 1);
   SetMenuItemText(theMenu, iEditUndoItem, (unsigned char *)itemString);
   DisableItem(theMenu, iEditUndoItem);
}

if (((theViewerState & kQ3ViewerHasModel) {
   EnableItem(theMenu, iEditCutItem);
   EnableItem(theMenu, iEditCopyItem);
   EnableItem(theMenu, iEditClearItem);
}
else {
   DisableItem(theMenu, iEditCutItem);
   DisableItem(theMenu, iEditCopyItem);
   DisableItem(theMenu, iEditClearItem);
}
```

(continued on next page)

```
/* Check that there's some data that we can paste. GetScrap returns a
   long that gives either the length of the requested type or a negative
   error code that indicates that no such type exists. */
tmpLong = GetScrap(nil, '3DMF', &theScrapOffset);
if (tmpLong < 0)
   DisableItem(theMenu, iEditPasteItem);
else
   EnableItem(theMenu, iEditPasteItem);
```

but first some conversion of color component values is necessary. While Macintosh Toolbox routines tend to work with the RGB system of specifying color, the QuickDraw 3D routines use an ARGB type that specifies an alpha channel component in addition to the red, green, and blue components. Conversion is necessary because each component of a QuickDraw 3D ARGB specification is a float in the range 0 through 1 rather than a 32-bit integer ranging from 0 through 65535 like the Macintosh Toolbox RGB components. See Listing 10 for the code that does the conversion.

Listing 10. Converting color component values

```
RGBColor        theRGBColor;
TQ3ColorARGB    theViewerBGColor;
...

Q3ViewerGetBackgroundColor(theViewer, &theViewerBGColor);
theRGBColor.red = theViewerBGColor.r * 65535.0;
theRGBColor.green = theViewerBGColor.g * 65535.0;
theRGBColor.blue = theViewerBGColor.b * 65535.0;

if (PickViewerBackgroundColor(&theRGBColor, "\pPick a viewer background
    color:")) {
   theViewerBGColor.a = 1;
   theViewerBGColor.r = theRGBColor.red / 65535.0;
   theViewerBGColor.g = theRGBColor.green / 65535.0;
   theViewerBGColor.b = theRGBColor.blue / 65535.0;
   Q3ViewerSetBackgroundColor(theViewer, &theViewerBGColor);
}
```

The routine PickViewerBackgroundColor, based on a routine described in the book *Advanced Color Imaging on the Mac OS*, uses the Macintosh Color Picker component to query the user for a new background color, returning a Boolean indicating whether the user chose a new color. This routine, shown in Listing 11, is a good deal simpler than it looks at first glance. We pass in the current background color and the prompt to be displayed in the Color Picker dialog. Since the Color Picker can use the Edit menu to support copy and pasting of color information, we need to tell it where our Edit menu is and which items in the menu are which. We then set up a Color Picker info structure, before calling PickColor (the guts of this routine). If the user cancels, we set the return value accordingly and return.

Listing 11. Letting the user choose a window background color

```c
Boolean PickViewerBackgroundColor(RGBColor *theRGBColor,
    unsigned char *thePrompt)
{
    ColorPickerInfo    cpInfo;
    Boolean            returnValue = false;

    /* Set the input color. */
    cpInfo.theColor.color.rgb.red = (*theRGBColor).red;
    cpInfo.theColor.color.rgb.blue = (*theRGBColor).blue;
    cpInfo.theColor.color.rgb.green = (*theRGBColor).green;

    cpInfo.theColor.profile = 0L;
    cpInfo.dstProfile = 0L;     /* No ColorSync destination profile */

    /* Set the Color Picker flags. */
    cpInfo.flags = AppIsColorSyncAware | CanModifyPalette |
                   CanAnimatePalette;

    /* Center dialog box on the deepest color screen. */
    cpInfo.placeWhere = kDeepestColorScreen;

    cpInfo.pickerType = 0L;     /* System default picker */

    /* Install event filter and color-changed functions. */
    cpInfo.eventProc = nil;
    cpInfo.colorProc = nil;
    cpInfo.colorProcData = 0L;

    /* Do a sanity check. */
    if (thePrompt[0] >= 255)
        thePrompt[0] = 255;

    BlockMove(thePrompt, cpInfo.prompt, thePrompt[0]);

    /* Describe the Edit menu for the Color Picker Manager. */
    cpInfo.mInfo.editMenuID = mEditMenu;
    cpInfo.mInfo.cutItem    = iEditCutItem;
    cpInfo.mInfo.copyItem   = iEditCopyItem;
    cpInfo.mInfo.pasteItem  = iEditPasteItem;
    cpInfo.mInfo.clearItem  = iEditClearItem;
    cpInfo.mInfo.undoItem   = iEditUndoItem;

    /* Display a dialog box to allow the user to choose a color. */
    if (PickColor(&cpInfo) == noErr && cpInfo.newColorChosen) {
        /* Use this new color. */
        (*theRGBColor).red = cpInfo.theColor.color.rgb.red;
        (*theRGBColor).blue = cpInfo.theColor.color.rgb.blue;
        (*theRGBColor).green = cpInfo.theColor.color.rgb.green;
        returnValue = cpInfo.newColorChosen;
    }
    return returnValue;
}
```

CHANGING THE RENDERER

QuickDraw 3D ships with two basic renderers: a wireframe and an interactive renderer, as illustrated by the examples in Figure 5.

Interactive renderer   Wireframe renderer
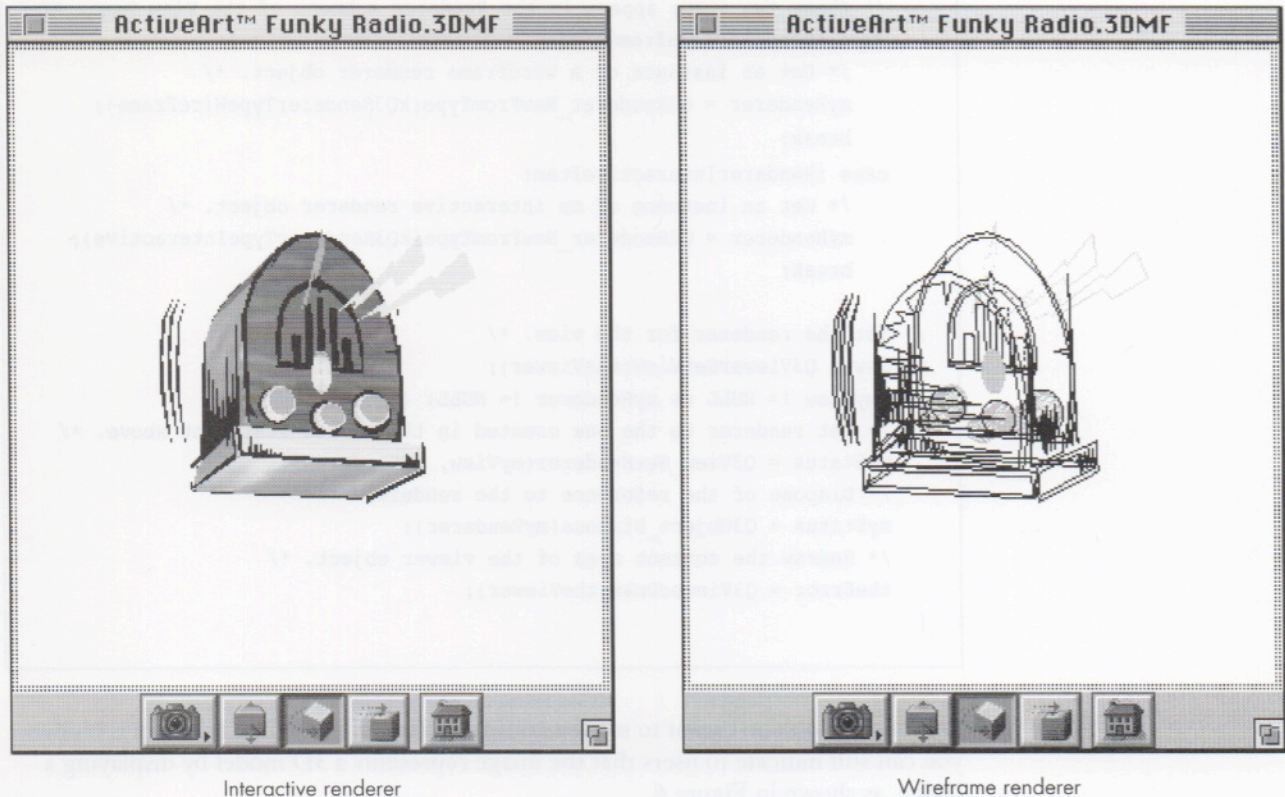
Figure 5. Drawing with the interactive and wireframe renderers

The Viewer shared library has no way to change the renderer, but we can use lower-level QuickDraw 3D routines to set the renderer and report the setting back to the user.

The renderer is associated with a view object, and we must have a view object in order to draw anything. The Viewer shared library contains a routine that enables us to get at the view, called Q3ViewerGetView.

Once we have the view object, we can start to extract information from it; in this case we'll need the renderer object associated with the view (see Listing 12).

HIDING AND SHOWING BUTTONS AND THE CONTROLLER STRIP

As mentioned earlier, you can control whether a button is displayed in the controller strip by toggling the appropriate flag. For example, to toggle whether the rotate button is displayed you can use the following code, which gets the viewer flags and bitwise-manipulates them:

```
theViewerFlags = Q3ViewerGetFlags(theViewer);
theViewerFlags ^= kQ3ViewerButtonOrbit;
Q3ViewerSetFlags(theViewer, theViewerFlags);
Q3ViewerDraw(theViewer);
```

You can display or hide other buttons in the same way by toggling the appropriate flag.

Listing 12. Setting the renderer

```
switch (menuItem) {
    /* These two items appear in the Renderer submenu of the View menu. */
    case iRendererWireframeItem:
        /* Get an instance of a wireframe renderer object. */
        myRenderer = Q3Renderer_NewFromType(kQ3RendererTypeWireFrame);
        break;
    case iRendererInteractiveItem:
        /* Get an instance of an interactive renderer object. */
        myRenderer = Q3Renderer_NewFromType(kQ3RendererTypeInteractive);
        break;
}
/* Set the renderer for the view. */
myView = Q3ViewerGetView(theViewer);
if (myView != NULL && myRenderer != NULL) {
    /* Set renderer to the one created in the switch statement above. */
    myStatus = Q3View_SetRenderer(myView, myRenderer);
    /* Dispose of the reference to the renderer. */
    myStatus = Q3Object_Dispose(myRenderer);
    /* Redraw the content area of the viewer object. */
    theError = Q3ViewerDraw(theViewer);
}
```

Sometimes you don't want to see the controller strip at all. When the strip is hidden, you can still indicate to users that the image represents a 3D model by displaying a badge, as shown in Figure 6.
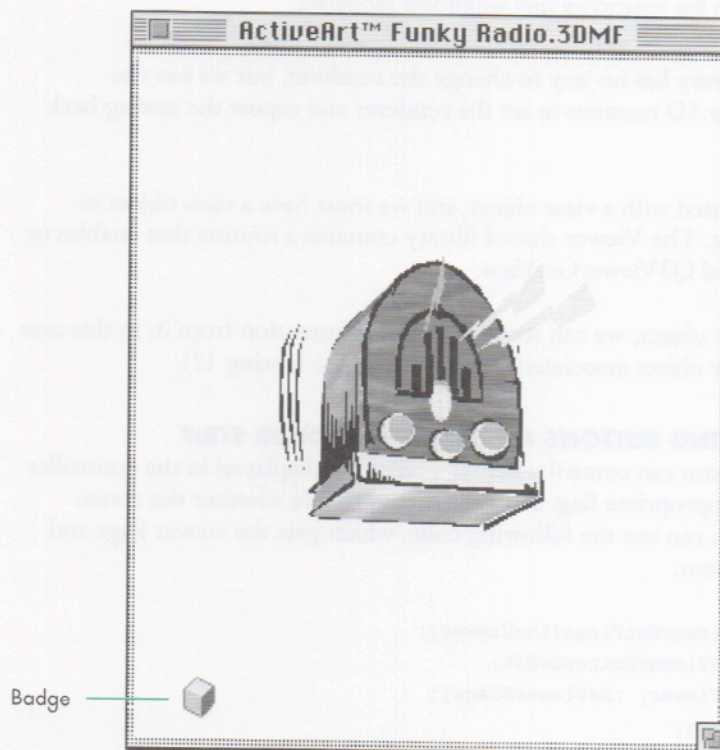

Badge

Figure 6. The 3D badge in a window with the controller strip hidden

The following code toggles the badge on and off:

```
theViewerFlags ^= kQ3ViewerShowBadge;
theViewerFlags ^= kQ3ViewerControllerVisible;

Q3ViewerSetFlags(theViewer, theViewerFlags);
Q3ViewerDraw(theViewer);
```

When the badge is displayed, the user can get the controller strip by clicking on the badge. The badge and the controller strip are mutually exclusive — if the badge is displayed, the controller strip should be hidden, and vice versa. In addition, badge control is one-directional for the user — the user can only switch from badge mode to controller strip mode. It's the responsibility of the application to redisplay the badge at appropriate times by setting the viewer object's kQ3ViewerShowBadge flag again and clearing the kQ3ViewerControllerVisible flag. For example, when a viewer object is deselected in a compound document, the application may switch the viewer object back to badge mode.

RESIZING THE VIEWER PANE WITHIN THE WINDOW

As mentioned earlier, the viewer pane can occupy the entire window or it can occupy just part of the window, as in a multimedia product. The code to draw the viewer pane smaller than the window uses the routine Q3ViewerSetBounds to define the bounds of the viewer object.

The code snippet in Listing 13 toggles the viewer pane between taking up the entire window and being inset a small amount. It keys off the kQ3ViewerDrawFrame flag; if this flag is set, the pane is inset.

Listing 13. Toggling the viewer pane between the entire window and just a part

```
theTmpRect = theWindow->portRect;

if (theViewerFlags & kQ3ViewerDrawFrame)
   Q3ViewerSetBounds(theViewer, &theTmpRect);
else {
   InsetRect(&theTmpRect, kInsetPixelsConst, kInsetPixelsConst);
   Q3ViewerSetBounds(theViewer, &theTmpRect);
}
theViewerFlags ^= kQ3ViewerDrawFrame;

GetPort(&savedPort);
SetPort((GrafPtr)theWindow);
EraseRect(&theWindow->portRect);
SetPort(savedPort);
```

Listing 14 shows how to resize the entire window. There are a couple of nuances here. We use the routine Q3ViewerGetMinimumDimension to calculate the minimum width and height of the window before resizing it with the routine SizeWindow. The minimum width is variable and depends on the number of buttons that are currently visible in the viewer. We also need to take into account the dimensions of the size box in the lower-right corner of the window. We can then set the bounds of the viewer object with the function Q3ViewerSetBounds.

Listing 14. Resizing the entire window

```
case inGrow:
    /* First we need to calculate the minimum size for this window.
       Fortunately, the Viewer library has a handy little utility
       function that we can use here. */
    theErr = Q3ViewerGetMinimumDimension(theViewer, &width, &height);
    growRect.top = height;
    growRect.left = width + 34;          /* +34 so the size box looks neat */
    growRect.bottom = kMaxHeight;
    growRect.right = kMaxWidth;
    newSize = GrowWindow(theWindow, theEventRecord.where, &growRect);
    if (newSize != 0) {
        width = LoWrd(newSize);
        height = HiWrd(newSize);
        SizeWindow(theWindow, width, height, true);
        Q3ViewerSetBounds(theViewer, &theWindow->portRect);
        Q3ViewerDraw(theViewer);
        DoDrawGrowIcon(theWindow);
    }
    break;
```

THE VIEW FROM HERE

Implementing the QuickDraw 3D Viewer in your application is an inexpensive way to get your feet wet before taking the plunge into QuickDraw 3D, as you've seen in this article. And remember — your application can mix and match QuickDraw 3D Viewer routines with QuickDraw 3D routines to extend the basic functionality of the Viewer. So go ahead and give your users a taste of 3D excitement. You may just decide that it's worth implementing QuickDraw 3D in full in your next application.

RELATED READING

- "QuickDraw 3D: A New Dimension for Macintosh Graphics" by Pablo Fernicola and Nick Thompson, *develop* Issue 22.

- *3D Graphics Programming With QuickDraw 3D* by Apple Computer, Inc. (Addison-Wesley, 1995).

- *Advanced Color Imaging on the Mac OS* by Apple Computer, Inc. (Addison-Wesley, 1995).

**TROY GAUL AND
VINCENT LO**

THE OPENDOC ROAD

Making the Most of Memory in OpenDoc

In Issue 28, we discussed how the OpenDoc Memory Manager works and how part editors manage Toolbox memory. This time we'll examine ways to use memory more efficiently in the OpenDoc environment.

We'll begin by talking about how to avoid memory leaks. Memory leaks, which can be a problem when developing traditional Macintosh applications, are as much a concern in OpenDoc. But because OpenDoc uses reference counting, there are a few extra things to pay attention to. We'll also discuss how to handle parameters correctly to avoid memory leaks, and we'll take a look at ways you can set up your part editor to maximize memory usage.

AVOIDING MEMORY LEAKS

OpenDoc objects and part editors use a reference-counting scheme that enables OpenDoc to keep track of which objects are in use. Each time a client acquires an object (through the object's Acquire method), the object's reference count is incremented by 1. When the object is no longer being used, the client releases it (by calling the object's Release method) and its reference count is decremented. The object's reference count indicates how many references to the object are being held by clients. When the reference count goes down to 0, the object can be destroyed without affecting any other objects. For more information on how reference counting works in OpenDoc, see the OpenDoc Road column in *develop* Issue 27, "Facilitating Part Editor Unloading."

If the acquired object doesn't get released when it should, the reference count doesn't go to 0 and the object remains in memory until the session ends. As a

result, a memory leak occurs because the occupied memory can't be used during the session.

To avoid reference count errors, it helps to keep in mind which classes are reference-counted and which methods affect an object's reference count. OpenDoc uses reference counting on classes whose objects often have more than one client. These classes are subclasses of ODRefCntObject, and many are classes that part editors interact with directly.

In general, if a method name starts with "Acquire," the reference count of the object named in the method is incremented when the method is called. When the object is no longer needed, the caller should release it. For example, if a part editor calls ODDraft::AcquireFrame to access a frame object, the reference count of the returned frame object is incremented. After the editor is done using the frame reference, a call to the object's Release method (ODFrame::Release) must be made to avoid a memory leak.

Some methods return a reference-counted object without affecting the object's reference count. These methods usually start with "Get." For example, ODFacet::GetFrame returns the frame object with which the facet is associated without incrementing the reference count of the frame object. In this case, the caller shouldn't call ODFrame::Release. Typically, a Get method is used to return an invariant or unchanged attribute of an object. In the case of ODFacet, the facet acquires and stores a reference to its ODFrame object. This reference isn't released until the ODFacet object is deleted. When ODFacet::GetFrame is called, ODFacet returns the stored reference to the caller. Since this reference remains valid until the ODFacet is deleted, you can use it as long as the ODFacet is a valid object. If you want to use the returned ODFrame object beyond the ODFacet's lifetime, you should call Acquire on the ODFrame to ensure that you have a valid reference to it.

The best way to avoid reference count errors is to familiarize yourself with the OpenDoc API and understand how it affects an object's reference count. The *OpenDoc Class Reference* provides a detailed description of reference counting for each method.

Temporary objects to the rescue. The code for acquiring a reference-counted object for a brief period of time and then releasing it turns out to be quite

TROY GAUL (tgaul@apple.com) recently joined the OpenDoc engineering team, where he's working with Java™. Having also written the sample part editor formerly known as Cappuccino, he has a caffeine buzz that should last into the next century. •

VINCENT LO (vincent@apple.com) is Apple's technical lead for OpenDoc. Since he recently introduced the OpenDoc team to Hong Kong cinema, it occasionally happens that the OpenDoc engineering meeting resembles a scene from a Hong Kong action movie. •

complicated. Listing 1 shows how complicated it can be to handle a reference-counted object when using exception-handling code.

Listing 1. Handling reference-counted objects

```
ODFrame* frame = kODNULL;
ODVolatile(frame);
// Make sure that the frame can be used in
// the CATCH block.
SOM_TRY
    // Acquire the frame.
    frame = draft->AcquireFrame(ev, id);
    // Do something with the frame here.
    ...
    // Release it when done.
    frame->Release(ev);
SOM_CATCH_ALL
    if (frame)
        frame->Release(ev);
SOM_ENDTRY
```

To help alleviate this problem, OpenDoc provides a utility library that uses stack-based C++ objects to wrap references to OpenDoc reference-counted objects. These C++ objects are called *temporary objects*. Whenever such a C++ object goes out of scope, its destructor is called and releases the reference-counted object.

The code fragment shown in Listing 2 does the same thing as the example in Listing 1 but uses temporary objects instead. This code is simpler and less error-prone.

Listing 2. Easier handling of reference-counted objects

```
SOM_TRY
    TempODFrame frame =
            draft->AcquireFrame(ev, id);
    // Do something with the frame here.
    ...
SOM_CATCH_ALL
SOM_ENDTRY
```

The OpenDoc utility library provides temporary objects for 17 reference-counted classes, including ODPart, ODFrame, ODExtension, and ODStorageUnit. For more information on creating temporary objects, see the "Temporary Objects" section of Appendix A in the *OpenDoc Cookbook*.

The OpenDoc utility library also provides temporary objects for objects that aren't reference-counted, such as ODByteArray and ODIText. These OpenDoc types deserve special attention in regard to memory usage.

The ODByteArray structure contains three fields: **_buffer**, **_maximum**, and **_length**. The **_buffer** field points to a memory block whose size is indicated by the **_maximum** field. **_length** is the number of bytes used; it has to be less than or equal to the value of **_maximum**.

Generally, ODByteArray is used instead of a raw pointer because the size of the memory block is included. This enables SOM and OpenDoc to pass data between processes without relying on shared memory. But because the **_buffer** field is hidden in the ODByteArray, the memory block can easily be forgotten. Failing to free this memory block when an ODByteArray is deallocated creates a memory leak.

The ODIText structure stores a user-visible string. One of its fields contains the string's format; the other is an ODByteArray that contains the text string. The memory block in the ODByteArray needs to be freed when the ODIText structure is deallocated.

Handling in and out parameters. Memory leaks can also occur when parameters aren't handled correctly. In an OpenDoc method, each parameter is designated as **in**, **out**, or **inout**.

- An **in** parameter passes data from the caller to the callee.

- An **out** parameter transfers data from the callee to the caller. A method's result also acts as an **out** parameter.

- An **inout** parameter passes data from the caller to the callee, which can then modify it and pass it back.

To determine a particular parameter's designation, you can check the ".idl" files, or see the *OpenDoc Class Reference* for detailed information on each parameter.

The parameter's designation defines the memory responsibility of the caller and callee. The part editor can use memory on the stack for parameters of primitive types or fixed-size data structures. But for strings, byte array buffers, and objects, the part editor must use the OpenDoc Memory Manager to do the following:

- allocate and deallocate memory for **in** and **inout** parameters passed to an OpenDoc object

- deallocate memory for **out** parameters returned from an OpenDoc object

- allocate memory for **out** parameters returned from the part editor's methods

If a part editor calls an OpenDoc method and doesn't deallocate the **out** parameter, the memory won't be freed until the session ends, causing a memory leak.

Since it's impossible to know how a piece of memory is allocated, OpenDoc and part editors have to use the OpenDoc Memory Manager as the common memory management facility. This is the only way to ensure that memory allocated by OpenDoc can be freed by the part editor and vice versa.

SETTING UP YOUR PART EDITOR

Because your part editor is used in documents with other part editors as users construct compound documents, it's important to make the best use of memory. Let's talk about some of the things you can do to minimize memory usage.

Keep the data section small. When creating a part (which is a shared library), the linker will generate code and data sections. The code section contains the instructions that make up your part editor. This section is read-only because it's file-mapped onto read-only memory when virtual memory is in use. The data section is stored separately because it needs to be writeable; it contains globals, static variables, transition vectors, virtual tables, and so on.

A single in-memory or memory-mapped copy of the code section is shared by all processes in which that code is used. The data section is handled differently: Each process instantiates a copy of the data section, making globals per-process rather than per-computer or per-part. Also, because there is normally one process per OpenDoc document, a separate data section usually exists for each document that contains a part bound to your editor. Therefore, you should control the size of your data section so that copies of it don't take up too much memory when multiple documents are open.

Here's what you can do to keep the size of your data section down:

- Limit your use of global variables — Since globals are stored in the data section, use them only for those things that must be per-process globals.

- Use read-only string constants — String constants that are writeable must be located in your data section because the compiler assumes that you might write into the memory associated with them. If you have the compiler make your string constants read-only (by checking the Make Strings ReadOnly box in the PPC Processor panel in CodeWarrior, for

example), these strings can be put into the code section instead of the data section. But remember that after doing this, you should not write to these string constants. You can still allocate memory in the OpenDoc heap for strings and write to them. You can also put string buffers, such as Str255 strings, on the stack in your code and write to them there. Note that any user-visible string constants should be stored in resources so that your editor can be localized.

- Avoid virtual functions — Space is made for virtual functions of C++ classes in the data section because the virtual tables must be written once (for each process) to point to the functions residing in the read-only code section. You can make the virtual table smaller by not making functions virtual. It's best to design your classes with as few virtual functions as possible, adding more only as the need arises.

- Reduce the number of transition vectors — For each import and export symbol in a library, there's a TVector, or transition vector. (CFM-68K calls them XVectors.) The TVector must be writeable because, like a virtual table, it has to be written once to point to the corresponding memory address when the code is loaded.

Reduce exports. By minimizing the number of symbols that are exported, you can save memory. Symbol name strings are stored in the PEF (Preferred Executable Format) container of your code fragment. If you're using a shared library that contains a framework or set of C++ classes, you usually need to export the symbols for each of the member functions in the shared library and import the relevant ones into your part editor to call them or subclass them. C++ functions have long symbol names because they include type signature information. As a result, the size of your code fragment can increase significantly.

This is particularly noticeable if you have multiple part editors that reference the same code, since they'll all have large tables of symbol names. You can use a tool like DumpPEF to check what type of information your code fragment contains and how much space it's taking up.

A workaround is to statically link classes to your part editor. This means fewer imports and less memory used. Of course, if you do this, you lose some of the advantages of sharing code via a shared library.

Package multiple part editors intelligently. If you're writing a suite of part editors for end users, it's a good idea to package them as separate editor files in the Editors folder. However, as mentioned earlier,

if you want to share common code, the amount of memory that's used by all the editors combined can be substantial.

Packaging all your part editors and the common code in a single code fragment reduces the number of imports and exports to almost nothing. But then you can't update just one of the editors in your suite — you have to replace the entire shared library. It's also detrimental if only one or two of your editors are being used, because the system loads the entire code fragment but only a portion of it is being referenced. This isn't as much of a problem if the user has virtual memory enabled, but without it, memory is wasted.

To combine multiple OpenDoc part editors into one code fragment, you have to compile the code for all of them together. You can do this either by putting them all into one project or by having multiple projects generate static library files and a master project that includes each of the single-part libraries. Then you need to make sure that the ClassData symbols for all the parts are exported as separate symbols, by using pragmas or a ".exp" file. Finally, you must include the 'nmap' resources from all the individual editors in the combined file. Of course, the IDs of these resources can't conflict, but since OpenDoc doesn't require any specific resource IDs for 'nmap' resources, that shouldn't be a problem to set up.

Use SOM classes instead of C++. If you'd like to separate framework code from part editor code (for example, to have multiple editors share the same framework or set of classes), note that there are several advantages to using SOM classes instead of C++ classes.

With SOM on the Mac OS, you only have to export the class's ClassData symbol. The virtual tables are maintained by the SOM kernel; they don't exist in your data section.

Since SOM has so little overhead, you can package multiple editors as separate code fragments (either in separate, replaceable files or in a single file). Editors that aren't being used won't be loaded.

You can also reduce the granularity of your shared libraries, such that different classes are in different shared libraries (again, in separate files or the same file). This allows you to split up your framework so that only the sections that the client needs are loaded into memory. For example, if you have one part editor that embeds other editors and another that doesn't, but they share the same framework, the framework's embedding code can be in a separate shared library from the code that's needed by all part editors.

SOM supports release-to-release binary compatibility, and it deals with the fragile base class problem of C++. It also defines a binary interface that supports languages other than C++. Currently, emitters on the Mac OS for C and C++ are available. Other languages can also be supported.

Don't be afraid to use SOM. Better tools for building SOM classes are being released. In particular, Direct-to-SOM support has been added to Metrowerks CodeWarrior's C++ compiler and Apple's MrCpp, so you can build SOM classes with much less effort and with a more familiar syntax.

Use #pragma internal. Space is also wasted when it's set aside for instructions and never used. By default, functions on a PowerPC™ processor are assumed to be external, so for the processor to jump to the routine, it's expected to go through a TVector. To do this, the compiler leaves room in the code for the linker to add the necessary instruction to restore the TOC (Table Of Contents) after a jump to a TVector. If it turns out that the routine is in the same code fragment, restoring the TOC isn't necessary, but because the space has already been inserted, the linker has little recourse but to put a no-op instruction in that place. (The code was generated expecting certain offsets, so the linker can't shuffle the code around easily.) Space is then wasted for calls that are internal to the code fragment.

There are a couple of ways around this. One thing you can do is to declare a function with the keyword **static** (this means that it can't be used outside the file it's defined in) so that the compiler can tell it's an internal function. You can also use the **internal** pragma in CodeWarrior. The following code marks the declaration of two functions as internal by enclosing them in a **#pragma internal** block. This informs the compiler that calls to those functions can be assumed to be internal calls, and it won't leave space for restoring the TOC after a TVector call.

```
#pragma internal on
    void InternalFunction();
    ODBoolean AnotherInternalFunction(short count);
#pragma internal reset
```

Note that a function internal to your code can still be an export from your shared library. In this case, your header should conditionalize its inclusion of **#pragma internal** for your own use so that external clients don't mistakenly see it as internal.

This technique is not used for CFM-68K code because calls there are assumed to be internal unless they're marked otherwise (with **#pragma import**). •

The MrPlus profiling tool can also be used to get rid of unneeded no-op instructions. You can get this tool and its documentation on the E.T.O. and MPW Pro CDs.

EVERY BYTE COUNTS

OpenDoc presents a new model for constructing software. However, many of the techniques you've used for the traditional application model can still be applied to the OpenDoc environment. By also incorporating some of the suggestions we've brought up here, you'll be able to further reduce your part editor's footprint and avoid memory leaks.

Gearing Up for Asia With the Text Services Manager and TSMTE

Are you eyeing Asian markets for your application? If so, the smartest thing you can do to gear up is to enlist the aid of the Text Services Manager (TSM), introduced with System 7.1 to help applications communicate with utilities that provide text services. Making your application TSM-aware, an easy matter if you're using TextEdit and TSMTE, will enable it to use the ser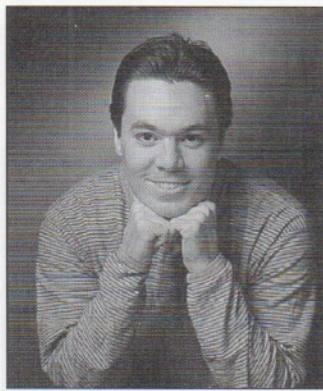vices of utilities designed to handle text input in Chinese, Japanese, and Korean. Your application will also be poised to take advantage of the wide variety of text services that eventually will be supported by the Text Services Manager.

TAGUE GRIFFITH

Localizing your application for Asian markets, or for Asian language–speaking customers in the United States, may seem like a daunting task to you, but take heart: the Text Services Manager (TSM) makes one aspect of localization, handling keyboard input, easier than you might imagine. Part of the WorldScript technology in the Macintosh Toolbox, the Text Services Manager enables applications and text service utilities to communicate without knowing anything about each other's internal structures or identities. When you make your application TSM-aware, you make it possible for your Asian language–speaking customers to use your application in concert with a utility program that does the necessary conversion of keyboard input.

This article shows you how to modify your TextEdit-based application to make it TSM-aware — that is, so that it makes the appropriate calls to the Text Services Manager. It doesn't take a lot of modifications, as you'll see from the sample application (called InlineInputSample) that accompanies this article on this issue's CD and *develop*'s Web site. Our application uses TSMTE, an extension that's shipped with the system), which extends TextEdit to handle the details of TSM awareness with minimal effort on the part of application writers. Using TSMTE should be sufficient for most applications; however, for intensive text-processing applications or applications using a different text-editing engine, you may need to handle all TSM processing yourself.

Before we look at the changes you need to make to your application to make it TSM-aware, I'll briefly explain how keyboard input works for Chinese, Japanese, and Korean. If you'd like to read more about common problems of localization, see "Writing Localizable Applications" in *develop* Issue 14. For details on the Text Services Manager, consult Chapter 7 of *Inside Macintosh: Text*.

TAGUE GRIFFITH (tague@apple.com) has appeared on stage with several famous rock bands and would like to marry a rock star when he grows up. In his spare time, Tague works in Apple's Text and International Engineering group. If anyone out there knows Courtney Love's e-mail address, please send it to Tague.•

ASIAN LANGUAGES AND KEYBOARD INPUT

As you can guess, supporting keyboard input for Asian languages isn't the same as handling English, because they're written in different scripts. A *script* is a writing system that can be used to represent one or more human languages.

English and other European languages are written in the Roman script, which is an *alphabetic* script. In alphabetic scripts, the various characters of the script are combined in different ways to form words. Alphabetic scripts have a small repertoire of characters compared to other types of writing systems. It's a simple matter to represent all the characters in an alphabetic script on a keyboard. Because there are fewer than 256 characters in such scripts, it takes only one byte to uniquely identify each character, so these scripts are known as *1-byte scripts*.

Asian languages are quite different, being written in scripts that include ideographic characters borrowed from ancient China. An *ideograph* is a symbolic character that usually represents a single concept, action, or thing. Figure 1 shows some examples of Japanese and simplified Chinese ideographs. Because each character represents a single concept, there are — by necessity — many, many more characters than in the Roman script. Most literate Chinese speakers know around 5000 ideographs, and a literate Japanese knows around 3000 ideographs. Two bytes are required to uniquely identify each character in an ideographic script, and thus these scripts are known as *2-byte scripts*. Chinese, Japanese, and Korean also incorporate alternative script systems based on syllabic or phonetic characters (characters that represent certain sounds).
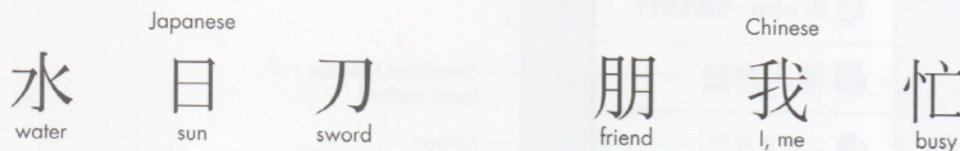
Japanese

水 日 刀
water sun sword

Chinese

朋 我 忙
friend I, me busy

Figure 1. Some Japanese and Chinese ideographs and their English translations

INPUT METHODS

How is it possible for users of 2-byte script systems to get by with a standard Macintosh keyboard? Obviously, they can't simply press the key corresponding to the one character they want out of 3000 or 5000 characters. Enter the text service utility known as an *input method* or a *front-end processor* (*FEP*). An input method allows users to type phonetic or syllabic characters on a standard keyboard and automatically converts what they type into ideographic representations.

For Chinese speakers, the appropriate input method converts keyboard input from Pinyin (Roman) or Zhuyinfuhao (phonetic, also known colloquially as Bopomofo) to ideographic Hanzi. For Japanese speakers, the input method converts input from phonetic Katakana or Hiragana into ideographic Kanji, as illustrated by the example in Figure 2. The input method for Korean speakers converts phonetic Jamo into nonideographic Hangul (complex clusters of Jamo).

Hiragana
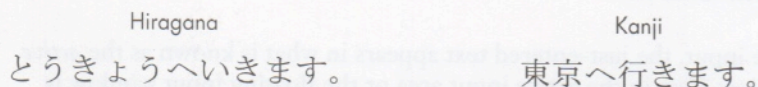
とうきょうへいきます。

Kanji

東京へ行きます。

Figure 2. The same sentence as entered in Hiragana and as converted to Kanji

Apple currently ships four 2-byte keyboard input methods: Kotoeri (Japanese), Power Input Method (Korean), Traditional Chinese (as used in Taiwan), and Simplified

Chinese (as used in the People's Republic of China). The same input methods are shipped with the Apple Language Kits, and third-party input methods are also available.

Regardless of the language, all input methods have a similar user interface. When more than one script is installed on the Mac OS, as is the case for localized systems since all systems have the Roman script installed, the Keyboard menu becomes available in the menu bar. Each available keyboard layout and input method is listed in the Keyboard menu; the icon for the active keyboard layout or input method appears as the menu's title in the menu bar. Figure 3 shows a Keyboard menu displaying items for Apple's Simplified Chinese and Kotoeri (Japanese) input methods, as well as keyboard layouts from some other script systems. The Simplified Chinese input method is active; it's checked in the menu and its icon appears highlighted in the menu bar. The pencil icon in the menu bar is displayed only when an input method is active (in other words, not when the user is typing in English or another language that doesn't require an input method); it's the title for the menu belonging to that input method. Some input methods use a different icon, but it appears in the same place as the pencil icon.
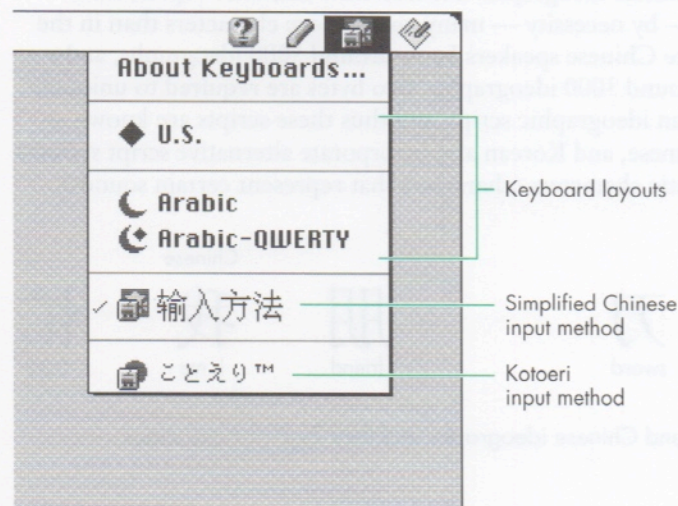

Figure 3. Input method icons in the Keyboard menu and the menu bar

BOTTOMLINE VS. INLINE INPUT
When the user begins typing, the raw text appears on the screen as entered, either in a *floating input window* that's usually displayed in the lower portion of the screen or in the application window where the text is intended to appear. The first style of text entry is known as *bottomline input*, while the second is called *inline input* (see Figure 4). Applications that aren't TSM-aware can make indirect use of the Text Services Manager's floating window service to enable bottomline input (as explained on page 7-13 of *Inside Macintosh: Text*), but users generally prefer inline input, which only TSM-aware applications can offer. TSM-aware applications can also offer bottomline input, which users may prefer if the size of the text displayed in the document makes reading the characters difficult.

In the case of inline input, the just-entered text appears in what is known as the *active input area* or *inline hole*. Text in the active input area or the floating input window is underlined in gray or highlighted in some other manner, depending on the application.

With either bottomline or inline input, the raw text is converted from its phonetic or syllabic representation to ideographic or complex syllabic characters, and the gray underline (if there is one) turns to black or changes in some other manner determined
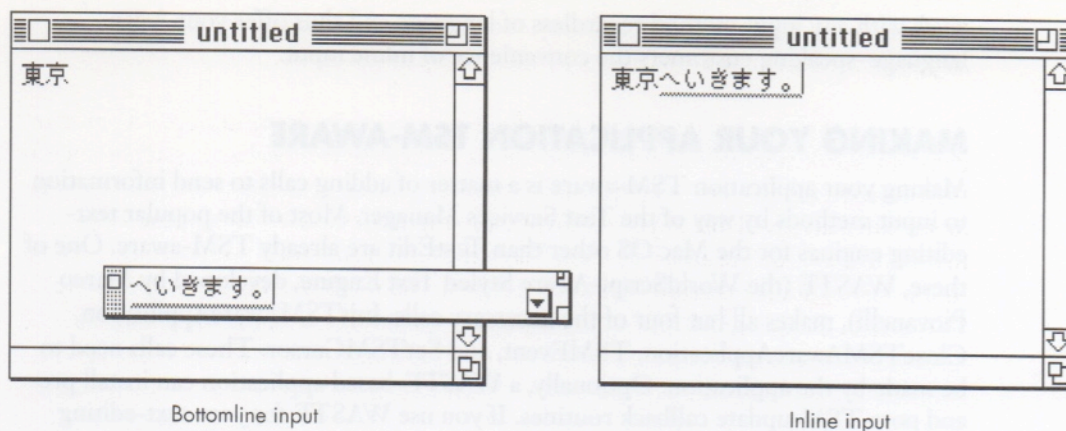
Figure 4. Bottomline vs. inline input

by the application, when the user gives a signal such as pressing the space bar after entering a sequence of characters. There may be more than one possible reading of a given character sequence, in which case the input method will display a list of candidates in a *candidate window*, as shown in Figure 5. When the user selects one of the candidate readings, the raw text is converted.
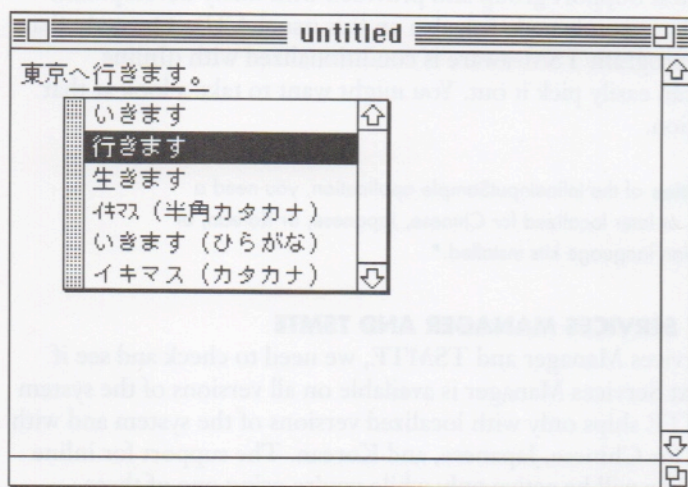

Figure 5. Selecting a conversion option for inline input in a candidate window

The user then confirms the converted text, generally by pressing Return. (In Korean, conversion happens continuously and automatically, and the text is confirmed when the user presses either Return or the space bar.) In the case of bottomline input, the confirmed text is flushed from the input window and sent to the application as key-down events. For inline input, the confirmed text is copied into the application's text buffer (as shown in Figure 5) and the active input area is closed. When the user begins typing again, the underline beneath the confirmed text disappears entirely and a new active input area opens.

Before you start feeling overwhelmed by all this, realize that most of the user interface elements I've just described are handled by the input method or TSMTE and not your application. The input method takes all the keystrokes and processes them; your application simply draws the input method's text buffer in the application window. All you need to do to get the benefit of this kind of text service is to make a few modifications to your application. Once your application is TSM-aware, you can

work with any input method regardless of language and thus offer your Asian language–speaking customers the convenience of inline input.

MAKING YOUR APPLICATION TSM-AWARE

Making your application TSM-aware is a matter of adding calls to send information to input methods by way of the Text Services Manager. Most of the popular text-editing engines for the Mac OS other than TextEdit are already TSM-aware. One of these, WASTE (the WorldScript-Aware Styled Text Engine, developed by Marco Piovanelli), makes all but four of the necessary calls: InitTSMAwareApplication, CloseTSMAwareApplication, TSMEvent, and SetTSMCursor. These calls need to be made by the application. Optionally, a WASTE-based application can install pre- and post-TSM-update callback routines. If you use WASTE for your text-editing engine, most of the techniques described in this section apply. The WASTE source code is available online at many popular Macintosh ftp sites; I highly recommend looking at it for examples of how to handle the TSM protocol directly.

Using TSMTE, as our application InlineInputSample does, you can make your TextEdit-based application TSM-aware with a few modifications to your event-handling, cursor-handling, window, and menu code. Most of the changes are quite simple and limited to particular subroutines of the application, as demonstrated by InlineInputSample. Our application is a version of TESample, a program written by Apple's Developer Technical Support group and provided with many development environments as part of the example code (it's also on this issue's CD). The code that makes our version of the program TSM-aware is conditionalized with qInline conditionals so that you can easily pick it out. You might want to take a look at that code as you read this section.

> **To see the full capabilities** of the InlineInputSample application, you need a Macintosh with System 7.1 or later localized for Chinese, Japanese, or Korean, or with one or more of the Asian language kits installed. •

TESTING FOR THE TEXT SERVICES MANAGER AND TSMTE

Before using the Text Services Manager and TSMTE, we need to check and see if they're available. The Text Services Manager is available on all versions of the system after 7.1. However, TSMTE ships only with localized versions of the system and with the Apple Language Kits for Chinese, Japanese, and Korean. The support for inline input discussed in this article will be active only while you're using one of these languages. Listing 1 shows the code we use to check for availability of the Text Services Manager and TSMTE. If we were writing our own protocol handlers, we would eliminate the gestaltTSMTEAttr test.

The selector gestaltTSMgrVersion returns the version number of the Text Services Manager if it's installed. You should test to make sure that the version is greater than or equal to 1, the current version of the Text Services Manager. This will allow your application to work with future TSM versions as well.

INITIALIZING THE APPLICATION

Once we've established that the Text Services Manager and TSMTE are available, we need to extend our Toolbox initialization sequence to initialize the Text Services Manager. This is done by calling InitTSMAwareApplication. We also want to store the current state of the Script Manager's smFontForce variable (the font force flag) and set it to false while our application is running. This flag ensures the correct text-handling behavior in applications that don't use the Script Manager. Since we're using the Script Manager to support text in different languages, we should turn this off, as shown in Listing 2.

Listing 1. Testing for TSM and TSMTE availability

```
static void CheckForTextServices(void)
{
    long  gestaltResponse;

    gHasTextServices = false;     // unless proven otherwise
    gHasTSMTE = false;            // unless proven otherwise

    if (TrapAvailable(_Gestalt)) {
        if ((Gestalt(gestaltTSMgrVersion, &gestaltResponse) == noErr) &&
              (gestaltResponse >= 1)) {
            gHasTextServices = true;
            if (Gestalt(gestaltTSMTEAttr, &gestaltResponse) == noErr)
                gHasTSMTE = BTst(gestaltResponse, gestaltTSMTEPresent);
        }
    }
}
```

Listing 2. Initializing as a TSM-aware application

```
if (!(gHasTSMTE && InitTSMAwareApplication() == noErr)) {
    // If this happens, just move on without text services.
    gHasTextServices = false;
    gHasTSMTE = false;
}
// Get global font force flag; make sure it's off whenever we run.
// Do this even if text services don't exist.
gSavedFontForce = GetScriptManagerVariable(smFontForce);
(void) SetScriptManagerVariable(smFontForce, 0);
```

Of course, since we do this work at initialization, we need to clean up when our application quits. In our termination routine, we restore the value of the font force flag and call CloseTSMAwareApplication. The font force flag also needs to be restored anytime control passes from the application to the system when we're dealing with fonts and such; it particularly should be restored in the case of a suspend event.

EXTENDING THE DOCUMENT STRUCTURE
Now we need to extend our document record to store the additional data structures related to TSM awareness. Our application's original DocumentRecord data structure is extended to include two additional fields, as follows:

```
typedef struct {
    WindowRecord    docWindow;
    TEHandle        docTE;
    ControlHandle   docVScroll;
    ControlHandle   docHScroll;
    TEClickLoopUPP  docClick;
    Boolean         modified;
    TSMTERecHandle  docTSMTERecHandle;   // added
    TSMDocumentID   docTSMDoc;           // added
} DocumentRecord, *DocumentPeek;
```

The TSMTERecHandle is used by TSMTE to maintain the data it uses. The TSMDocumentID identifies a *TSM document*, which is an opaque data structure used by the Text Services Manager to maintain the current status of the input methods in use. Generally, one TSM document (defined by the TSMDocument data type) is allocated per application window, but some applications may allocate a single global TSM document. Since the TSM document maintains context/state information about the current input method, you should customize allocation of the TSM document for your application's text-handling behavior.

Each TSM document also maintains a reference constant, which can be set by the application. Applications using TSMTE must stuff the TSMTERecHandle into the refCon; if you're creating your own handlers, the refCon can be customized to suit the needs of your application.

CREATING AND DELETING A TSM DOCUMENT

When creating a user document, we need to set up the TSM document and the TSMTERecHandle correctly, as illustrated in Listing 3. We pass four parameters to NewTSMDocument. The first parameter indicates the version of the TSM protocol that we're using; currently, the only defined protocol version is 1. The next parameter, supportedInterfaces, is an array of OSTypes that the Text Services Manager uses to find components that support the service the client is interested in. For our application, we set supportedInterfaces[0] to kTSMTEInterfaceType, indicating that we're using TSMTE support. Other applications providing full TSM support will want to set supportedInterfaces[0] to kTextService. Currently these are the only defined interface types. The final two parameters are a pointer to the TSMDocumentID storage and a pointer to our TSM refCon (in this case the TSMTERecHandle).

Once we've allocated the TSM document, we need to set up the TSMTERecHandle. We set the TextEdit record for the handle and install UniversalProcPtrs for the pre- and post-update handlers. These handlers are called before and after TSMTE handles the update event. The pre-update handler in our sample application works around a bug in TSMTE version 1.0, and the post-update handler adjusts the scroll bar to bring the new text into view. We also set the updateFlag and the refCon.

Obviously, since we allocate certain structures when we create our TSM document, we need to deallocate those structures before we destroy the TSM document (that is, when we close the window). Listing 4 shows how to handle deleting a TSM document.

Listing 3. Creating a TSM document

```
if (good && gHasTSMTE) {
    supportedInterfaces[0] = kTSMTEInterfaceType;
    if (NewTSMDocument(1, supportedInterfaces, &doc->docTSMDoc,
            (long) &doc->docTSMTERecHandle) == noErr) {
        TSMTERecPtr tsmteRecPtr = *(doc->docTSMTERecHandle);
        tsmteRecPtr->textH = doc->docTE;
        tsmteRecPtr->preUpdateProc = gTSMTEPreUpdateUPP;
        tsmteRecPtr->postUpdateProc = gTSMTEPostUpdateUPP;
        tsmteRecPtr->updateFlag = kTSMTEAutoScroll;
        tsmteRecPtr->refCon = (long) window;
    }
    else
        good = false;
}
```

Listing 4. Deleting a TSM document

```
if (theDocument->docTSMDoc != nil) {
    (void) FixTSMDocument(theDocument->docTSMDoc);
    // DeleteTSMDocument might cause a crash if we don't deactivate
    // first, so...
    (void) DeactivateTSMDocument(theDocument->docTSMDoc);
    (void) DeleteTSMDocument(theDocument->docTSMDoc);
}
```

The FixTSMDocument call causes the Text Services Manager to confirm the text in the active input area and enter it into the user document. In a real application you'd then give the user the opportunity to save the user document before deleting the TSM document and closing the window, but we skip that step here. After "fixing" the document, we call DeactivateTSMDocument and then DeleteTSMDocument. We need to deactivate the document before deleting it because in certain circumstances DeleteTSMDocument may crash if we call it on an active document.

MODIFYING THE EVENT LOOP

Making your application TSM-aware with TSMTE requires very little modification to your existing event loop code. First we need to give the Text Services Manager an opportunity to handle events that might actually be for an input method and not our application. This is accomplished by calling TSMEvent and passing in the EventRecord returned from either WaitNextEvent or GetNextEvent. In our sample application, we wrap the call to TSMEvent in the IntlTSMEvent subroutine, as shown in Listing 5, to work around a bug that could cause the port to be set to the wrong window. Listing 6 shows how we modified the application's event loop to call TSMEvent before handling an event.

Listing 5. The IntlTSMEvent subroutine

```
static Boolean IntlTSMEvent(EventRecord *event)
{
    short       oldFont;
    ScriptCode  keyboardScript;

    // Make sure we have a port and it's not the Window Manager port.
    if (qd.thePort != nil && FrontWindow() != nil) {
        oldFont = qd.thePort->txFont;
        keyboardScript = GetScriptManagerVariable(smKeyScript);
        if (FontToScript(oldFont) != keyboardScript)
            TextFont(GetScriptVariable(keyboardScript, smScriptAppFond));
    }
    return TSMEvent(event);
}
```

Next, input methods must be given a chance to handle mouse events for the pencil menu. Whenever the user presses the mouse button in the menu bar, we need to pass the menu selection to the Text Services Manager and see if it's a system menu before we try to handle it as an application menu. We modify the subroutine for handling mouse-down events as shown in Listing 7.

After calling our subroutine to adjust the menus so that the invalid items are dimmed, we pass the menu selection to the Text Services Manager. We call TSMMenuSelect with the menu result as a parameter. If the selection is in the pencil menu, the Text Services Manager will handle it for your application and return true; otherwise, it will return false and your application should handle the selection normally. Regardless of

Listing 7. Checking the menu selection in the mouse-down event handling

```
case inMenuBar:            // Process a mouse-down in menu bar (if any).
   AdjustMenus();
   menuResult = MenuSelect(event->where);
   if (!(gHasTextServices && TSMMenuSelect(menuResult)))
      DoMenuCommand(menuResult);
   HiliteMenu(0);           // Needed even if TSM or Script Manager handles
                            // the menu.
   break;
```

the results of TSMMenuSelect, your application needs to call HiliteMenu(0) to finish handling the selection.

The next step in handling a menu selection is to confirm the text in the active input area with a call to FixTSMDocument before performing the menu action. This is recommended in the current Macintosh Human Interface Guidelines. But this guideline is somewhat contested, so if you don't think it makes sense for your application, you might consider a more selective policy for automatically confirming inline input. Whatever you decide, remember to be consistent and try it out on real users.

Now that we've finished modifying our menu-handling code, we'll move on to modifying our window event handler, as shown in Listing 8. We need to make sure that the active TSM document is changed when necessary. Assuming that we're allocating one TSM document per user document or window, we also need to handle window activate and deactivate events differently. When we receive a deactivate event, we need to call DeactivateTSMDocument on the TSM document associated with that window. Similarly, we need to call ActivateTSMDocument when we receive an activate event for our window.

Listing 8. Handling window events

```
void DoActivate(WindowPtr window, Boolean becomingActive)
{
   RgnHandle     tempRgn, clipRgn;
   Rect          growRect;
   DocumentPeek  doc;

   if (IsDocumentWindow(window)) {
      doc = (DocumentPeek) window;
      if (becomingActive) {
         ...   // TextEdit-handling code
         if (doc->docTSMDoc != nil)
            (void) ActivateTSMDocument(doc->docTSMDoc);
      }
      else {
         if (doc->docTSMDoc != nil)
            (void) DeactivateTSMDocument(doc->docTSMDoc);
         ...   // TextEdit-handling code
      }
   }
}
```

If you use your own scheme for allocating TSM documents, be sure to call ActivateTSMDocument and DeactivateTSMDocument when appropriate. If you don't activate TSM documents correctly, the system can get confused and revert to Roman, in which case the user may click on a run of 2-byte text and not get inline input. If you're debugging your program and this happens, check to see what the current keyboard script is by calling

```
keyScript = GetScriptMgrVariable(keyScript);
```

If the keyboard script is smRoman, the problem could be related to activating and deactivating TSM documents.

We finish up our event loop modifications by ensuring that our application handles mouse-moved events in a way that works with the Text Services Manager and input methods. Input methods need to track mouse-moved events within the content area of your application, because when the user moves the cursor over an active input area, the input method needs to change the cursor's shape. This is demonstrated in the first half of Listing 9. Before attempting to change the cursor's appearance, your application should call SetTSMCursor, which returns a Boolean indicating whether the input method has already changed the cursor's appearance.

Listing 9. Allowing the input method to change the cursor

```
// Before we commit to anything, let's check whether some text service
// has a different idea.
if (!(gHasTextServices && SetTSMCursor(mouse))) {
    // Change the cursor and the region parameter.
    if (PtInRgn(mouse, iBeamRgn)) {
        SetCursor(*GetCursor(iBeamCursor));
        CopyRgn(iBeamRgn, region);
    }
    else {
        SetCursor(&qd.arrow);
        CopyRgn(arrowRgn, region);
    }
}
// No matter how nice the region, with text services it can't be bigger
// than a point. Yes, this defeats the purpose of all the calculations.
if (gHasTextServices)
    SetRectRgn(region, mouse.h, mouse.v, mouse.h, mouse.v);
```

Now we need to set up the correct mouseRgn parameter to WaitNextEvent. To work correctly with input methods, our application should create a mouseRgn that's no larger than a point, as shown in the second half of Listing 9. If we don't set up this region correctly, the input method may not be able to interact with our users as they would expect. Yes, being forced to use a point-sized region sort of defeats the purpose of a mouseRgn. We're working on extending the Text Services Manager so that you can obtain the region that the input method is interested in, but for now you'll just have to live with this limitation.

ADDING FONT-KEYBOARD SYNCHRONIZATION

Both TextEdit and WASTE automatically perform font-keyboard synchronization, meaning that they adjust the Keyboard menu so that the current keyboard layout or

input method reflects the user's font selection. For instance, if the user selects Chicago or Courier (Roman fonts) from the Font menu, the application sets the current keyboard to a Roman keyboard layout (U.S., German, Italian, or whatever the default Roman keyboard is, according to the localizer). Similarly, if the user selects Osaka or HonMincho (Japanese fonts), the keyboard is *synchronized* with the font selection.

If you're not using TextEdit or WASTE, you can add the code in Listing 10 to provide font-keyboard synchronization. Using Toolbox routines, we determine the script of the font from the font ID. Then we call KeyScript, which changes the Keyboard menu settings to the default keyboard and input method combination for the new script.

Listing 10. Synchronizing the keyboard with the font selection

```
case mFont:
    GetMenuItemText(GetMenuHandle(mFont), menuItem, theFontName);
    GetFNum(theFontName, &theFontID);
    theTextStyle.tsFont = theFontID;
    TESetStyle(doFont, &theTextStyle, true, te);
    if ((*te)->selEnd - (*te)->selStart > 0)
        theDocument->modified = true;
    AdjustScrollbars(window, false);
    theScript = FontToScript(theFontID);
    KeyScript(theScript);
    break;
```

Font-keyboard synchronization is another one of those contested international human interface issues. Japanese users tend to be very divided on the issue, since this feature makes it difficult to set Roman characters to the Osaka font (the workaround is to select the Roman mode within the input method after selecting the Osaka font); however, users working in most other 2-byte languages like font-keyboard synchronization. Again, the best idea is to try out this interface feature on several real users and get their impressions.

OTHER USER INTERFACE ISSUES

When you make your application TSM-aware, you may need to work around a couple of other user interface issues. One of them is implementing passwords. When the user is typing a password, you don't want an input method to come up, so you should change the current keyboard layout to the Roman default. This is accomplished similarly to font-keyboard synchronization, using the following code:

```
KeyScript(smRoman);              // Switch to Roman.
KeyScript(smKeyDisableKybdSwitch); // Lock out keyboard switching.
... // Do your password stuff here.
KeyScript(smKeyEnableKeyboards);
KeyScript(smKeySwapScript);
```

If you're writing game software, in play mode the user can't or shouldn't be manipulating input with an input method. In this case, you should also change the current keyboard layout to Roman, and you might want to lock out keyboard switching. In certain other modes (such as the high score list), users do need the services of an input method, so you should restore the original keyboard layout.

Also, even though most users prefer inline input, some actually don't like it, or they prefer the bottomline style of input in certain situations. For example, it's difficult to distinguish Kanji characters at point sizes smaller than 12, so when the user is working on a document in a small font, looking at the raw input in a larger font in the floating input window may be desirable. You should provide a preference mechanism so that the user can select either inline or bottomline input. This is implemented by calling the TSM function UseInputWindow and then passing it the document ID and a Boolean indicating whether the user wants inline input.

TESTING YOUR APPLICATION

Once you've made your application TSM-aware, you'll want to test it with a variety of input methods, since not all of them interpret the TSM protocol in quite the same way. If you're fortunate enough to have testers who speak Chinese, Japanese, or Korean, you've got it made. But if you don't have anyone around who speaks these languages, it's still possible to test your application in English with an input method. For some suggestions on how non–Asian language speakers can successfully test your code, see "Testing Two-Byte Script Support."

THE TSM PROTOCOL

Although TSMTE should provide enough support for most application developers who want to make their TextEdit-based applications TSM-aware, some developers may want or need to handle all the details themselves. This section provides a very brief overview of the TSM 1.0 protocol for the benefit of the latter.

The TSM protocol is based on the exchange of Apple events between the application and the input method, allowing them to share information about the active input area. The protocol consists of a suite of three required events (Position To Offset, Offset To Position, and Update Active Input Area) and an optional fourth event (Get Text).

THE POSITION TO OFFSET EVENT

The Position To Offset event is used to convert a global position into a meaningful offset in the document's text buffer. This event is sent from the input method to the application when the input method handles an event that occurs within the context of the application content. (Remember that you need to call TSMEvent to give the Text Services Manager a chance to handle mouse and keyboard events.)

The application must respond to this event by extracting a point on the screen from the Apple event and converting that point to an offset in the text of the particular document. In the TSM protocol, offsets are defined as long integers. The application returns the offset as a key in the reply event.

THE OFFSET TO POSITION EVENT

The Offset To Position event is sent by the input method when it needs to determine the global position of a particular document offset. When responding to this event, the application must return a global point corresponding to this offset. Optionally, the application can also return information about the typographical style and orientation (vertical vs. horizontal) of the text. Each of these elements is returned as a key in the reply event.

THE UPDATE ACTIVE INPUT AREA EVENT

The Update Active Input Area event is used by the input method to ask the application to update the active input area. The event contains an offset range array that breaks the updated text into particular ranges. The offset range is accompanied

TESTING TWO-BYTE SCRIPT SUPPORT
BY GREG ANDERSON

I know what you're thinking: "Great, now I can add support for 2-byte scripts to my application. But I can't read Japanese or Chinese characters, so how am I supposed to test my code to make sure it does the right thing with 2-byte characters?"

Good news: You don't have to learn Japanese or Chinese. Instead, you can test your code using the 2-byte Roman input mode provided by Kotoeri. That's right — I said "2-byte Roman."

Most of the characters in the ASCII character set are replicated in one section of the Japanese 2-byte character set; you'll find the numbers and letters around $8250 to $829A. The 2-byte Roman characters look like their 1-byte counterparts, except that they're always monospaced and they take about twice as much space horizontally as the 1-byte characters do. These characters are very useful for testing 2-byte script support, because they look the same as the characters you use every day but they behave like other 2-byte characters such as Chinese and Japanese ideographs.

Entering the 2-byte Roman characters is easy. After you've installed the Japanese Language Kit, choose Kotoeri from the Keyboard menu (shown earlier in Figure 3) and click the button labeled with the wide uppercase A in the operations palette, which will then look like this:

Then just type Roman characters the same way you normally would. For example, if you want a Q, hold down the Shift key and press the Q key on the keyboard. You're now ready to edit 2-byte Roman characters in your application to make sure it handles them correctly. Here are some things to check:

- Is the insertion point drawn in the right place, or are you using a Roman font to measure Japanese text?

- If the insertion point is positioned after a 2-byte character and you press the Delete key, is the entire character deleted, or do you leave the first byte dangling in your document?

- If you add a 2-byte uppercase *B* (which has an encoded value of $8261) to your document and use your Find command to search for a 1-byte lowercase *a* (which has an encoded value of $61), does your application find the 2-byte *B*? (It shouldn't, and it won't if you're using CharacterByteType correctly.)

If these things work for you, you're well on your way to supporting 2-byte scripts in your application.

by a highlight range, which indicates how the application should highlight the particular range of text.

THE GET TEXT EVENT

Get Text is an optional event used only by Kotoeri, the Japanese input method, and some third-party Japanese input methods. It's documented in a March 1994 technical note by Takayuki Mizuno, available only in Japanese and roughly entitled "Kotoeri's Private Apple Event, Get Text." It's an extension to the TSM protocol developed by Apple as a way for an input method to retrieve text that has already been confirmed. Since most localized Japanese applications support this event, your users will probably expect it from your application as well.

TSMTE provides support for the Get Text event, so if you use the techniques in this article your application will be able to take advantage of this extension. The Get Text event is defined in Table 1. Listing 11 demonstrates how to handle the event directly.

TSM NOW AND FOREVER

Although the changes required to make your TextEdit-based application TSM-aware are small, they dramatically overhaul the experience available to Asian language–speaking users. Try compiling our sample application both with and without the

Table 1. Definition of the Get Text event

Event class	kTextServiceClass		
Event ID	kGetText (= 'gtxt')		
Requested action	Returns the current selection as the return parameter		

	Keyword	**Descriptor type**	**Data**
Required parameters	keyDocumentRefcon	typeLongInteger	Standard refCon parameter
	keyAEServerInstance	typeComponentInstance	Standard component instance parameter
Optional parameters	keyAEBufferSize (= 'buff')	typeLongInteger	Maximum number of bytes the input method can receive
Return parameters	keyAETheData	typeText	The text specified by the current selection. The maximum byte length is specified by the keyAEBufferSize parameter. Any portion of the text that exceeds the buffer length

Listing 11. Handling the Get Text event

```
pascal OSErr HandleGetText(const AppleEvent *theAppleEvent,
                   const AppleEvent *reply, long handlerRefcon)
{
#pragma unused (handlerRefcon)

    OSErr            err;
    ComponentInstance serverInstance;
    TSMTERecHandle    docRefcon;
    Handle            text;
    Size             textLen;

    // Identify event and get document refCon and server instance.
    err = IdentifyTSMCallback(theAppleEvent, &docRefcon,
                            &serverInstance);
    if (err) return err;

    // Get selected text from document.
    HLock((Handle) docRefcon);
    err = DoGetText(*docRefcon, serverInstance, &text, &textLen);
    HUnlock((Handle) docRefcon);
    if (err) return err;

    // Add selected text as return parameter.
    HLock(text);
    err = AEPutParamPtr(reply, keyAETheData, typeText, (Ptr) *text,
                        textLen);
    DisposeHandle(text);

    return err;
}
```

(continued on next page)

Listing 11. Handling the Get Text Apple event *(continued)*

```
OSErr DoGetText(TSMTERecPtr tsmteRecPtr, ComponentInstance
                serverInstance, Handle *text, Size *textLen)
{
#pragma unused (serverInstance)

    TEHandle hTE;
    short    selStart, selEnd;

    hTE = tsmteRecPtr->textH;
    selStart = (*hTE)->selStart;
    selEnd   = (*hTE)->selEnd;
    *textLen = selEnd - selStart;
    *text = NewHandleClear(*textLen);
    HLock((Handle) *text);
    BlockMove((Ptr) (*((*hTE)->hText) + selStart), (Ptr) (**text),
            *textLen);
    HUnlock((Handle) *text);
    return noErr;
}
```

inline support flag set so that you can see the difference it makes to users. Even if you don't have plans to ship your application localized for Chinese, Japanese, or Korean, making it TSM-aware will please your users who have an Apple Language Kit installed and want to use your application with their non-English data.

Besides having an immediate payoff, the work you do to make your application TSM-aware will have a future payoff as well. While version 1.0 of the Text Services Manager offers support solely for keyboard input methods, future versions will be part of the framework for supporting handwriting and speech/dictation input methods as well as more general text services such as interactive spelling checkers and intelligent document scanners. By making your application TSM-aware now, you'll be poised to take advantage of the wide variety of services that will be available with TSM 2.0.

RELATED READING

- "Writing Localizable Applications" by Joseph Ternasky and Bryan K. "Beaker" Ressler, *develop* Issue 14.

- *Guide to Macintosh Software Localization* by Apple Computer, Inc. (Addison-Wesley, 1992).

- *Inside Macintosh: Text* by Apple Computer, Inc. (Addison-Wesley, 1993), Chapter 7, "Text Services Manager."

- Technotes TE 27, "Inline Input for TextEdit With TSMTE," and OV 20, "Internationalization Checklist."

- *Understanding Japanese Information Processing* by Ken Lunde (O'Reilly & Associates, 1993).

- *Writing Systems of the World* by Akira Nakanishi (English edition, Charles E. Tuttle, 1980).

- IDRIS scripts, http://idris.com/scripts/Scripts.html.
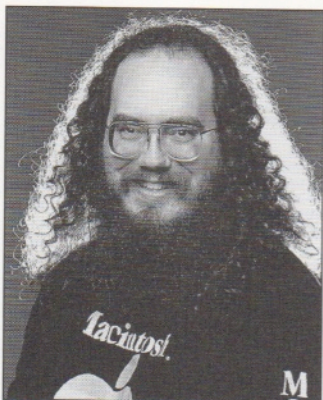
DAVE POLASCHEK

PRINT HINTS

Sending PostScript Files to a LaserWriter

A question that popped up pretty often for me when I worked in Apple's Developer Technical Support group, and that continues to appear in the Mac programming newsgroups on the Internet, is "How do I send PostScript™ files to a LaserWriter?" The simplest answer appears to be to use the PostScriptHandle picture comment. Wrong! In the Print Hints column "The Top 10 Printing Crimes Revisited" in *develop* Issue 26, I said that this would be a misuse of the PostScriptHandle picture comment, but I didn't give a full explanation of just how to send files to a printer. This column will explain how to send complete PostScript files the correct way.

First, though, I'd like to backtrack a little and explain more thoroughly why you shouldn't use the picture comment to send complete PostScript files. After all, if you do implement code that uses this technique, the files get to the printer, pages come out, and things seem mostly to work. What's the problem?

WHY NOT USE POSTSCRIPTHANDLE?

The problem with using the PostScriptHandle picture comment to send files to a printer is that you're using the Printing Manager to do some of the work for you and then bypassing it unexpectedly. The problems that will show up when you do this may not be obvious, but when a user does notice them it can lead to confusion.

First, when you use the PostScriptHandle picture comment, the LaserWriter driver has already set up the PostScript state for QuickDraw imaging. It has changed the coordinate system and has sent down the **md** dictionary that it will need to draw pages. At best, this is extra baggage that your PostScript files don't

need in the way. At worst, since the driver has changed the coordinate system, your PostScript file may not print correctly. There's also a chance that the extra memory used by the LaserWriter's PostScript dictionary may cause your job not to print at all.

Second, using PostScriptHandle is wasteful. If background printing is enabled, the complete PostScript file, plus the extra things you don't need, will be spooled to the user's hard drive. Since what you're interested in is just getting the file to the printer, this is wasteful. In some cases, the job won't even be able to print, since your user won't have enough free space on the hard drive to hold a second copy of the PostScript file.

Finally, there's an aesthetic problem. The Printing Manager counts the pages that are going to be sent by looking at how many times PrOpenPage is called during your print job. If you've got a multiple-page PostScript file, the page count displayed by the Printing Manager will be out of whack. While messages like "Printing Page 4 of 1" won't actually hurt the user, spreading confusion is bad.

THE RIGHT WAY(S)

There are actually two right ways to send PostScript files to the printer. They're essentially the same in that you open a connection directly to the printer and send it the file you want to print, but the implementations are very different. You can either use classic networking and the PAPWorkStation.o library, or you can use Open Transport, which contains support for the PAP protocol. (For more on PAP, see "PAP? What's That?")

The classic networking solution has the advantage that it's available in all Macintosh computers, out of the box. But it's more difficult to implement. Since you're working at a lower level, there's more room for you to get things wrong — but there's also less worry of having library code that doesn't do things the way you want. Note that your code (at least part of it) can't be PowerPC native, because the libraries are 680x0 only.

Open Transport is a much simpler way to implement sending PostScript files to your printer. You talk to the networking library at a higher level, and you get a PowerPC-native implementation of networking. But of course you lose compatibility with older Macintosh models.

DAVE POLASCHEK (davep@best.com, http://www.best.com/ ~davep/) now works for LaserMaster Corp., where he does "Mac things" that confuse the people who actually build printers, since Dave seems to spend most of his time designing spiffy icons and then dragging them around his screen. In his spare time, Dave scrapes ice off his windshield, jump-starts cars, and de-ices locks. Earlier this year he cut a hole in a lake and spent hours sitting on the ice waiting for fish to find his hook (while he consumed a judicious amount of antifreeze). Who said life in Minnesota isn't exciting?•

PAP? WHAT'S THAT?

The Printer Access Protocol (PAP) is the protocol spoken by all LaserWriters (and third-party Macintosh-compatible PostScript printers). It's designed around a few simple ideas:

- Only one computer can be talking to the printer at once, so there's a unique connection between the two.

- When the printer is ready to receive some data, it will ask the computer for that data. Similarly, when the printer wants to send data back to the computer, the computer must have already asked for that data. In other words, a read must always be active.

- There's a separate status channel for the printer to report things like "Printer On Fire" or "Paper Jam."

This last point means that there are actually multiple connections open for the one connection to the printer. This is a major source of the complexity of the code in the classic networking case. Open Transport hides this complexity from you.

You can find out more about PAP in Chapter 10 of *Inside AppleTalk*.

We'll look at both these solutions; sample code for each one accompanies this column on this issue's CD and *develop*'s Web site. The choice of which technique to use is up to you. Note also that there may be other, more attractive alternatives in the future. For now, however, these are the two best options.

The sample code for the classic networking solution is the SendPS tool — an MPW tool rather than a full application, but still useful for demonstrating how things work. The process is also described in the *MacTutor* article "Laser Print DA for PostScript." If you're interested in the nitty-gritty details, consulting the code and that article is your best bet. Here I'm just going to give an overview of the code. In a few cases, I'll make suggestions for how you might enhance the code to make a real-world application.

The main thing you need to understand about the PAP library that Apple supplies for classic networking is that it reports the connection status to you via a few variables rather than by way of completion functions. It's not truly asynchronous code, so if you want to write a program that will run happily in the background, sending PostScript files to a printer, you've got some extra work to do. It's possible, but a little tricky.

When we're in the process of opening a connection (in the code near the call to PAPOpen in sendps.c), we look at the **wstate** variable to tell us what's happening. All these state variables have three basic states: a positive value means we're waiting for the printer; negative values are errors; and 0 means the operation we're watching has completed. We also call PAPStatus periodically to get the printer's status so that we can display it to the user.

Once the connection is opened, we issue a PAPRead call. This is necessary because when the printer wants

to talk to us, it can't just tell us it's ready to talk, but rather we must have asked it for some data first. (Remember, PAP is a protocol where the parties have to ask for data.) We check the **rstate** variable periodically to see if the printer has said anything to us. When it does, we need to read the data it has sent us and issue another PAPRead call.

Now we're ready to start sending blocks of data. We issue a PAPWrite call, and for each call, we watch the **wstate** variable to see when the write is done. While we're waiting for the writes to complete, we need to remember to check the printer's status and display it to the user (our write might be "stuck" because the printer is out of paper and can't print a page, for example; the user should see this so that she can replenish the paper).

Once we've sent all our data, we send the end-of-file to the printer. We do this by sending a packet with no data, but with the **eof** flag set. If we've got another job to send, there's no need to close the connection and reopen it; we can just start sending it as soon as the printer acknowledges our end-of-file.

That's the quick overview of the SendPS tool. If you need more guidance, see the source code; I've tried to comment it so that everything will make sense.

THE OPEN TRANSPORT CODE
The main code for the Open Transport solution is considerably easier to follow than the classic networking code, primarily because you don't have to worry about

multiple state variables in order to send. You simply create PAP and PAPStatus endpoints (connections), and the main loop calls the Snd and Rcv functions for the PAP endpoint. There's a callback function that gets notified when the state of the endpoint changes and then sets a single state variable to match the state of the endpoint. Exception handling can be localized to this callback, which makes the code easier to read, and the status reporting can be handled in one place as well.

Because we don't need to continually poll the state variables, it's much easier to fit this code into an application, and the application will cooperate better with other applications. Getting good performance is also quite a bit easier with the Open Transport–based code than with the classic networking code, since you don't have to worry about waiting in the wrong place, polling a state variable.

WRAPPING UP

Implementing code to send PostScript files directly to a printer is more difficult than using the PostScriptHandle picture comment, but it offers a number of benefits to your users. The process is more straightforward, no extra print dialogs need to be shown, and no extra storage is required for spool files. You can provide as much or as little status information as you like (for example, you could add progress bars showing how much of the job is completed). And performance should improve in most cases, since you'll be talking directly to the printer, rather than through the layer of the Printing Manager. Convinced? I should hope so.

RECOMMENDED READING

- *Inside Macintosh: Open Transport* and *Open Transport Client Developer Note* (Apple Computer, Inc., 1996), available on the Open Transport Web site, http://www.devworld.apple.com/dev/opentransport/.

- *Inside Macintosh: Networking* by Apple Computer, Inc. (Addison-Wesley, 1994) and *Inside AppleTalk*, 2nd ed., by Apple Computer, Inc. (Addison-Wesley, 1990).

- Macintosh Technical Q&A QD 35, "Determining the Selected Printer's Address."

- "Laser Print DA for PostScript" by Mike Schuster, *MacTutor* Vol. 2, No. 2 (February 1986). Articles from *MacTutor*, since renamed *MacTech Magazine*, can be found at http://web.xplain.com/mactech.com/magazine/features/articlearchives.html.

- *Lost Beauties of the English Language*, by Charles Mackay, LL.D. (London: Bibliophile Books, 1987). Originally published by Chatto & Windus in 1874.

- *The UNIX-Haters Handbook* by Simson Garfinkel, Daniel Weise, and Steven Strassman (IDG Books, 1994).

High-Performance ACGIs in C

Asynchronous Common Gateway Interface (ACGI) programs allow Macintosh HTTP servers to do external processing tasks ranging from custom HTML forms processing to controlling hardware devices. ACGIs are usually written in AppleScript (which limits them to handling only one server request at a time). High-performance ACGIs, ones that are capable of handling multiple simultaneous requests, need to be written in a high-level language like C. The resulting ACGI will work with any HTTP server that supports the WebSTAR WWWΩ Apple event suite.

Now that you've got your HTTP server up and running on your Macintosh, people are flocking to your Web site by the thousands. The only problem is that you've written all of your Asynchronous Common Gateway Interface programs (ACGIs) in AppleScript and their performance is leaving much to be desired. You know you should be writing your ACGIs in C for speed, but you think that will be a lot of work.

Well, have I got news for you! A full-blown, multithreaded, high-performance ACGI program for use with Macintosh HTTP servers is easier to write than you think. If you've worked through one of the introductory Macintosh programming books, you already know just about everything you need to.

When all is said and done, an ACGI is little more than a simple, Apple event–aware application that knows how to process Apple events in threads. Most of the work is concentrated in decoding the Apple event parameters that make up each server request. Hopefully you won't feel so overwhelmed by ACGIs written in C (or any other high-level language) after you've read this article, and you can get on with using them to hot-rod your Web site!

I've made writing an ACGI easier for you by providing a generic ACGI program, which accompanies this article on this issue's CD and *develop*'s Web site. I designed the program (which I'll be referring to as an ACGI "shell") in such a way that you can create your own ACGIs just by customizing a handful of routines. The messy details of accepting multiple requests from an HTTP server, and then handling each request in its own thread of execution, are taken care of for you. The program even relieves

KEN URQUHART

KEN URQUHART received his Ph.D. in physics in 1989 and has been dividing his time between physics and computer science ever since. Ken's work has taken him and his wife from North America to Japan and back again. Their cats (who travel with them wherever they go) have been extremely good sports about international travel. Ken's pretty sure the cats understand English perfectly well — they're simply choosing to ignore him unless they want food, body heat, or the litter box cleaned. •

you of the burden of URL-decoding the post and search arguments (including breaking up all of the *name=value* pairs and translating them from the ISO-8859 Latin-1 character encoding used by most browsers into the standard Macintosh Roman encoding).

I've also provided a rich set of convenience routines that perform the following tasks:

- give you easy access to all the arguments and parameters that make up a server request
- help you compose your HTML replies
- get and set various ACGI performance-tuning parameters
- allow you to gently turn away new requests when your ACGI is very busy
- gracefully shut down the ACGI if the need arises

I've tried to provide enough support to make it possible for you to forget most of the details of interacting with an HTTP server and concentrate on writing the code needed to implement your custom form processing.

The ACGI shell program, compiled under CodeWarrior as a PowerPC application with no optimizations, takes up a little under 42K on disk (not including custom code that you must add to process your requests). Memory requirements are dictated by the number of concurrent requests you want to handle and how much stack space you allocate to each running thread. In a typical case, the shell should provide uniform response to about five to ten concurrent requests in a 1 MB memory footprint.

WHAT'S AN ACGI?

Before I can tell you what an ACGI is, I need to explain what a CGI is. This requires a bit of background on what HTTP servers are all about.

WHAT'S A CGI?

HTTP servers are designed to do one thing and to do it very well: respond to requests from Web browsers. If the request is for a file that resides somewhere in the server's directory tree, the server locates the file, reads its contents, and then sends the information back to the browser. Other requests such as image map or form processing are handed off to auxiliary programs that communicate with the server by using the Common Gateway Interface (CGI) protocol. When the server receives a request that must be handled by a CGI program, the server starts up the CGI (if it wasn't already running) and passes it the request. The CGI is responsible for parsing and decoding the request parameters, processing them, and then composing the HTML response. The server takes care of returning the response to the requesting browser.

Being a computer program, a CGI can readily interact with databases, transaction processing systems, or even connected serial devices to process a given request. So CGIs allow your Web site to serve up a wide variety of dynamic information.

The structure of a CGI program is dictated by the HTTP server and by the operating system. The first Macintosh HTTP server was MacHTTP, written by Chuck Shotton. He used Apple events for server/CGI communication and defined a special event suite (WWWΩ) for this purpose. He later extended this suite, adding several more parameters, when he wrote WebSTAR — the commercial version of MacHTTP. His suite has become the de facto standard for server/CGI interaction on the Macintosh. As such, you can be sure that most other Macintosh HTTP servers will support it.

WebSTAR-like servers use custom Apple events to communicate with CGIs and can
call them either synchronously or asynchronously.

- Synchronous calls require the server to suspend processing while it waits for
  the Apple event reply from the CGI.

- Asynchronous calls allow the server to send the request to the CGI and then
  continue processing other connections while the CGI does its work.

Asynchronous calls are almost always preferable for a popular Web site that's
receiving several connection requests a second.

SO *NOW* WILL YOU TELL ME WHAT AN ACGI IS?

An ACGI is a CGI that's called asynchronously by the HTTP server (you're surprised
to hear this?). Furthermore, when an ACGI is written to handle each request in a
separate thread of execution (enabling it to deal with multiple requests simultaneously),
it's referred to as a *threaded ACGI.*

To write a threaded ACGI for the Macintosh, you need to understand the following:

- how Web browsers send CGI requests to HTTP servers

- how a Macintosh HTTP server uses the WWWΩ Apple event suite to pass
  these requests along to an ACGI

- how an ACGI can arrange to process each Apple event in a separate thread
  of execution

- how to extract the URL-encoded data from the Apple events so that the
  ACGI can process it

While it would be just about impossible to describe each of these points in detail in
one short article, I do provide brief overviews as I talk about the functions of the
ACGI shell.

Like other threaded ACGI solutions (described in "Other Techniques for Developing
a Threaded ACGI"), my technique uses cooperative threads as opposed to
preemptive threads. This allows you to call any Toolbox routine you want when
you're carrying out your form processing. Preemptive threads currently have many
Toolbox calling restrictions (see the article "Concurrent Programming With the
Thread Manager" in *develop* Issue 17).

THE STRUCTURE OF THE ACGI SHELL

Just as there are many ways of writing a Macintosh application, there are many ways
to write an ACGI shell. I've taken the simplest possible approach and avoided using
an application framework like MacApp or PowerPlant. My ACGI shell is written in
plain C and consists of three logically separate code sections:

- a main program that receives Apple event requests from an HTTP server and processes them in separate threads of execution
- the set of customizable request-processing routines
- a set of convenience routines that simplify accessing the request data, composing HTML response pages, and controlling the runtime behavior of the ACGI

The code is split into two source files (acgi.c and www.c), two include files (acgi.h and www.h), and one resource file (acgi.rsrc). The main application and the convenience routines are located in acgi.c, while the routines that you'll need to customize are in www.c. The include file acgi.h contains the public prototypes for the convenience functions you can call from www.c, while the include file www.h contains the function prototypes and data structure definitions used by routines in both source files.

THE ROUTINES YOU NEED TO CUSTOMIZE

The file www.c contains six routines that you'll need to customize to implement your own custom form processing. Four routines are called exactly once by the main program while the ACGI is running. A fifth routine is called at idle time in the main event loop, while the last one is called to process each HTTP request.

WWWGETLOGNAME

When the ACGI starts up, one of the first things the main program does is open a log file to write progress messages to. It gets the name of the file by calling this routine:

```
char *WWWGetLogName(void);
```

Customizing WWWGetLogName allows you to specify the name of the log file. All you typically need to do is write something like this:

```
char *WWWGetLogName(void)
{
    return "acgi.log";
}
```

The one gotcha here is that I've used ANSI file I/O routines to simplify the program code. So you must always be sure to return a valid ANSI filename (a plain filename fewer than 31 characters long with no full or partial Macintosh file path prepended to it). Note that some Macintosh ANSI libraries will allow filenames prefixed by partial paths as long as the total length of the string is no longer than 255 characters.

WWWGETHTMLPAGES

After the log file is opened, the main program will ask you to build four HTML error pages that are returned to the HTTP server when one of these general errors occurs:

- The ACGI is declining (refusing) to process requests.

- The ACGI is too busy to handle a new request.

- The ACGI has run out of memory.

- The ACGI has run into an unexpected problem.

The routine you use to construct your pages is as follows:

```
void WWWGetHTMLPages(Handle refused, Handle tooBusy, Handle noMemory,
   Handle unexpectedError);
```

The main program passes in four handles. Each handle contains a standard HTTP response header, and you're responsible for appending whatever HTML text you want for the error pages. This allows you to control the "look and feel" of the error messages returned by your ACGI. Perhaps the simplest approach here is to put the HTML error pages into text files located in the same directory as your ACGI and then append them to the handles with the convenience routine HTMLAppendFile:

```
void WWWGetHTMLPages(Handle refused, Handle tooBusy, Handle noMemory,
      Handle unexpectedError
{
   HTMLAppendFile(refused, "acgiRefused.html");
   HTMLAppendFile(tooBusy, "acgiTooBusy.html");
   HTMLAppendFile(noMemory, "acgiNoMemory.html");
   HTMLAppendFile(unexpectedError, "acgiUnexpected.html");
}
```

Other convenience routines allow you to read the text from string and text resources, so you have some flexibility here. The idea behind WWWGetHTMLPages is to allow you to create your HTML error pages early in the initialization phase so that they'll always be available for use.

WWWINIT

After the main program has completed its initialization steps, you're given a chance to carry out any private initialization you need to do before beginning form processing. This might include calling the ACGI runtime-tuning routines, initializing your own global variables, reading resources into memory, building HTML template pages, or opening connections to external databases and other computers. The prototype is

```
OSErr WWWInit(void);
```

If you run into problems during your initialization, simply return a nonzero code. The main program checks the return code and immediately quits to the Finder when the code is nonzero.

If you have no special initialization to do, you could write this routine as follows:

```
OSErr WWWInit(void)
{
    return (noErr);
}
```

WWWQUIT

When the main program exits its main event loop, it calls this next routine to give you one last chance to clean up after yourself (close files, database connections, and so on):

```
void WWWQuit(void);
```

If you don't need to do any cleaning up, you can write something as simple as this:

```
void WWWQuit(void) { }
```

WWWPERIODICTASK

The main program allows you to carry out idle-time processing by calling the following routine at the end of each pass through the main event loop:

```
OSErr WWWPeriodicTask(void);
```

This is where you'd place code to check that connections to other computers are still alive or carry out any background processing initiated by previous server requests. If you have no idle-time processing, you could write the following:

```
OSErr WWWPeriodicTask(void)
{
    return (noErr);
}
```

The main program checks the return code from this routine and, if the code is nonzero, quits to the Finder (after trying to gracefully abort all currently running threads).

WWWPROCESS

The last routine you must customize is the one that processes a server request:

```
OSErr WWWProcess(WWWRequest request);
```

When the HTTP server sends the ACGI a request through an Apple event, the main program creates a new thread and passes the Apple event data into the thread. The thread extracts the request data from the Apple event and packs it into a private data structure. The thread then calls WWWProcess, passing a pointer to the private data structure in the **request** parameter. You extract information from the data structure with the convenience routines (described later).

If you need to abort the processing of a request, you can return one of the four error codes errWWWRefused, errWWWTooBusy, errWWWNoMemory, and errWWWUnexpected. These cause the corresponding HTML error pages that you built in the routine WWWGetHTMLPages to be returned to the server.

THE MAIN PROGRAM

As mentioned previously, the main program is a simple Macintosh application — simpler than most of the programs described in introductory Macintosh programming books. It's important to remember that an ACGI is meant to interact with HTTP

servers, not live users. It doesn't need any windows, complex menus, or even an About box. Its purpose in life is to respond to Apple events and not mouse clicks or keystrokes.

Furthermore, you cannot assume that a human will always be watching the server screen, ready to react to dialog boxes or alerts. If an ACGI runs into trouble, it should try to recover as best it can and keep going. For example, if a required external database shuts down, an ACGI might return an "out of service" response to each request until the database comes back online. If an ACGI runs out of memory, it might simply quit and allow the HTTP server to launch a fresh copy of it the next time a request comes in. Hopefully, that would cure the problem in the short term.

An efficient, low-overhead ACGI is therefore a windowless, Apple event–aware program that posts no alerts or dialogs. It implements only the Apple and File menus. For simplicity, the About item in the Apple menu does nothing except show the name of the ACGI (although there's nothing to stop you from implementing an About box if you want to). The File menu contains the single item Quit. A log file is used to record all informational, error, and debugging messages.

As shown in Listing 1, the main program starts by calling ACGIInit to set itself up. Then it runs the main event loop, calling ACGIEvent to process each new event, until the global gDone flag is set and all threads have completed. The program then cleans up after itself by calling ACGIQuit.

THREADS AND THE MAIN EVENT LOOP

The presence of threads affects the main event loop shown in Listing 1 in three ways. First, the loop doesn't exit as long as there are active threads. This ensures that all threads processing HTTP server requests complete their work before the ACGI shuts down. Second, there are two different sleep times for WaitNextEvent: gThreadSleep when threads are running and gIdleSleep when they're not. We need idle time to give the threads a chance to run. This means we should use a rather small value for **sleep** when gThreads is greater than 0. On the other hand, when there are no outstanding requests, we should set **sleep** to a large value to avoid wasting CPU time. The exception to this rule is when you have periodic tasks, in which case you should call ACGISetSleeps in WWWInit to set gIdleSleep to get the idle time you need.

Third, there's the inner loop that repeatedly calls YieldToAnyThread. This routine causes the Thread Manager to turn control over to the oldest running thread. This thread keeps control until it too calls YieldToAnyThread to turn control over to the next running thread. This continues until the newest thread calls YieldToAnyThread and control returns to the main event loop (see "Concurrent Programming With the Thread Manager" in *develop* Issue 17).

It's important to call YieldToAnyThread frequently inside your request-processing code, usually after you complete a logical step in your processing and no less than every 1 to 2 ticks of the Macintosh clock (1 tick = 1/60th of a second). Don't bother putting your calls to YieldToAnyThread inside a timed loop as we did in the main event loop. Just call it often throughout your code: it's a very low overhead call. The secret to uniform response time to all requests is not to allow any one thread to hog the CPU.

YieldToAnyThread is enclosed in a timed loop to give threads enough time to do useful work when running on a Power Macintosh. Currently, there's a context switch from native PowerPC mode to 680x0 emulation mode when WaitNextEvent is called. In addition, historical reasons guarantee that WaitNextEvent always waits at least 1 tick before it returns. Calling YieldToAnyThread only once per pass through the main event loop means that threads would get time only once every 1/60th of a second and

```
Listing 1. The ACGI main program

// Include files and function prototypes
...

static Boolean      gDone = false;
static unsigned long gThreads = 0;
static long         gThreadSleep = 4;
static long         gIdleSleep = 0x7FFFFFFF;
static long         gWNEDelta = 8;

void main(void)
{
    EventRecord    theEvent;
    long           sleep;
    unsigned long  nextWNE;

    ACGIInit();
    while (!gDone || gThreads > 0) {
        if (gThreads > 0)
            sleep = gThreadSleep;
        else
            sleep = gIdleSleep;
        if (WaitNextEvent(everyEvent, &theEvent, sleep, nil))
            ACGIEvent(&theEvent);
        nextWNE = TickCount() + gWNEDelta;
        do {
            YieldToAnyThread();
        } while (TickCount() <= nextWNE);
        ACGIPeriodicTask();
    }
    ACGIQuit();
}
```

a lot of useful CPU time would be wasted in mode switches. The timed loop could result in a thousandfold performance increase — without noticeably affecting other applications — for ACGIs running compute-bound threads that frequently yielded.

THE INITIALIZATION ROUTINE ACGIINIT

ACGIInit carries out seven distinct steps to get the ACGI going:

1. Initialize the Toolbox.

2. Get the name of the log file by calling WWWGetLogName and then open it.

3. Check to see that both Apple events and the Thread Manager are present.

4. Set up the menu bar.

5. Install the Apple event handlers.

6. Call WWWGetHTMLPages to build the four generic HTML error pages.

7. Call WWWInit to initialize your processing environment.

If ACGIInit runs into trouble, it calls ACGIFatal to write an error message to the log file and quit. If you run into trouble in WWWInit you should write a meaningful

error message to the log with ACGILog and return a nonzero result code. ACGIInit will write the code to the log and then quit.

THE LOGGING ROUTINES ACGILOG AND ACGIFATAL
Two routines that write zero-terminated strings to the log — ACGILog and ACGIFatal — are shown in Listing 2. In these routines, gLog is an ANSI **FILE*** variable that's local to the source file acgi.c. It points to the open log file.

Listing 2. Logging routines

```
void ACGILog(char *msg)
{
    DateTimeRec dt;
    ThreadID    theThread;

    if (gLog == NULL)
        return;
    GetTime(&dt);
    GetCurrentThread(&theThread);
    fprintf(gLog, "%4d/%02d/%02d\t%02d:%02d:%02d\t%010lu\t%s\n", dt.year,
        dt.month, dt.day, dt.hour, dt.minute, dt.second, theThread, msg);
    fflush(gLog);
}

void ACGIFatal(char *reason)
{
    if (gLog != NULL) {
        ACGILog(reason);
        ACGILog("That was a fatal error...shut down.");
    }
    ExitToShell();
}
```

ACGILog prefixes each message with the date and time and the ID number of the thread it was called in. The items are tab-separated so that you can later import the log into a spreadsheet and sort it by date, time, or thread ID. This can be useful when you're trying to debug an ACGI or gather statistics based on the messages you wrote into the log during processing. ACGIFatal calls ACGILog to write its message to the log and then quits the program immediately without waiting for running threads to complete. It's meant to be called only from within ACGIInit.

PERIODIC TASKS AND THE TERMINATION ROUTINE
ACGIPeriodicTask runs periodic tasks by calling your WWWPeriodicTask routine and then checking for a nonzero result code (in which case it writes the code to the log and, if the code is positive, sets gDone to true). The termination routine ACGIQuit is the last routine called by the main program. It shuts down processing by calling your WWWQuit routine and then closes the log.

EVENT HANDLING IN THE MAIN EVENT LOOP
Since an ACGI is basically a simple Macintosh application with no windows, no About box, and only the Apple menu and File menu (which supports the single item Quit), you don't have to worry about activate and update events, and suspend/resume events

only need to set the cursor to an arrow. Keystrokes are important only if they're Command-key equivalents that might represent a menu selection. This limited event handling is carried out entirely in the routine ACGIEvent and its small support routine DoMenu (for menu and Command-key handling). ACGILog is used to report any errors that are encountered.

ACGIEvent doesn't need to do any special processing at this level to handle threaded Apple events. It just calls AEProcessAppleEvent like any other application. Details of the threading process are hidden away in the Apple event handler that's called in response to HTTP server requests.

APPLE EVENT SUPPORT IN THE ACGI

The ACGI must support the four core Apple events and the custom event sent by HTTP servers and must be able to process HTTP events in threads. Here are the details of how the ACGI shell implements the required Apple events and the threading of the server requests.

SUPPORTING CORE APPLE EVENTS

Any application that supports Apple events must support the four core events (Open Application, Open Document, Print Document, and Quit Application), as well as any custom Apple events needed for communication with other programs. Because the ACGI doesn't have any documents, doesn't do any printing, and does all the application initialization before accepting the first Apple event, it can deal with the four core events with the single handler HandleAECore:

```
#define kQuitCoreEvent  1
#define kOtherCoreEvent 0
static pascal OSErr HandleAECore(AppleEvent *event, AppleEvent *reply,
    long refCon)
{
    if (refCon == kQuitCoreEvent)
        gDone = true;
    return (noErr);
}
```

The ACGI sets the handler reference constant, refCon, to kOtherCoreEvent for the 'oapp', 'odoc', and 'pdoc' events and to kQuitCoreEvent for the 'quit' event. When the handler is called, it simply returns noErr if the refCon is kOtherCoreEvent and sets gDone to true if the refCon is kQuitCoreEvent.

THREADING HTTP SERVER REQUESTS

The WWWΩ Apple event class defines a single event ID ('sdoc') to pass requests to ACGI programs. This is the event that the ACGI shell responds to. To handle multiple server requests at once, the ACGI must process each request in its own thread of execution.

This leads to some complications in the code because the Apple Event Manager was designed to have only one event active at any given time. To process multiple Apple events in threads, the ACGI will have to suspend each new Apple event in the main thread of execution, put each suspended event into its own thread for processing, and then let each thread resume its suspended Apple event at the end of processing so that replies are returned to the HTTP server.

The one catch here is that when an event is suspended, the pointers to the event and reply data structures become invalid. The ACGI must therefore make copies of the

event and reply data structures (and not just the pointers) before suspending an event. These copies of the AEDescs are passed into the thread for processing.

So, the processing flow for threading HTTP server requests is as follows:

1. ACGIInit makes HandleSDOC the handler for HTTP server requests.

2. The main event loop (running in the main thread) receives an HTTP server request and calls AEProcessAppleEvent as usual.

3. HandleSDOC (also running in the main thread) receives the Apple event.

4. If there are too many threads running or the ACGI is refusing connections, the handler immediately returns an HTML page indicating that the server request cannot be processed. Otherwise, the handler allocates a handle called **params** to hold copies of the Apple event and its reply. Note that the complete data structures must be copied, not just the pointers to them, because the pointers become invalid when the event is suspended.

5. HandleSDOC creates a new thread and passes **params** into it. If the thread cannot be created, **params** is disposed of and the error code is returned.

6. HandleSDOC increments the count of running threads and then suspends the current Apple event and returns. The main event loop is now free to accept another server request.

7. The main event loop regains control and calls YieldToAnyThread almost immediately. Each processing thread is given time to run, and control eventually passes to the new thread.

8. The new thread begins life by calling SDOCThread. This routine makes local copies of the suspended Apple event and its reply and then disposes of the **params** handle that was passed to it by HandleSDOC.

9. SDOCThread extracts parameters from the Apple event, URL-decodes them, and then calls WWWProcess to process the server request. WWWProcess calls YieldToAnyThread frequently to give time to other threads and to allow the main thread to accept new Apple events. When WWWProcess finishes, it returns a handle containing the HTML response page.

10. SDOCThread places the response into its copy of the Apple event reply and then resumes execution of the suspended event. The event in this thread is now considered complete. You're guaranteed that no other Apple event will be "current" at this time because HandleSDOC suspends each new event before any of the processing threads are given time to run.

11. The thread decrements the global counter gThreads and then returns (causing the thread to be disposed of).

With this processing flow as a guide, the associated code practically writes itself. The HandleSDOC routine is shown in Listing 3.

Listing 3. Handling HTTP server requests

```
static unsigned long gMaxThreads = 10;
static Boolean       gRefusing = false;
static long          gThreadStackSize = 0;
static ThreadOptions gThreadOptions = kCreateIfNeeded | kFPUNotNeeded;
```

Listing 3. Handling HTTP server requests *(continued)*

```
typedef struct AEParams {
   AppleEvent  event;
   AppleEvent  reply;
} AEParams;

void SDOCThread(void *threadParam);
OSErr ACGIReturnHandle(AppleEvent *reply, Handle h);

pascal OSErr HandleSDOC(AppleEvent *event, AppleEvent *reply,
   long refCon)
{
   AEParams**  params;
   ThreadID    newThreadID;
   OSErr       err;

   // [1]   Too many threads already running?
   if (gThreads >= gMaxThreads)
      return (ACGIReturnHandle(reply, gHTMLTooBusy));

   // [2]   Should we handle this request?
   if (gDone || gRefusing)
      return (ACGIReturnHandle(reply, gHTMLRefused));

   // [3]   OK to run...make copies of event and reply.
   params = (AEParams**) NewHandle(sizeof(AEParams));
   if (params == nil)
      return (errAEEventNotHandled);
   (*params)->event = *event;    // Copy the data structures...
   (*params)->reply = *reply;    // ...not just the pointers to them!

   // [4]   Create the thread, passing in the copies of event and reply.
   err = NewThread(kCooperativeThread, (ThreadEntryProcPtr) SDOCThread,
         (void*) params, gThreadStackSize, gThreadOptions, nil,
         &newThreadID);
   if (err != noErr) {
      DisposeHandle((Handle) params);
      return (err);
   }

   // [5]   Increment the count of running threads and then suspend
   //       the current event so that we can accept new events.
   gThreads++;
   return (AESuspendTheCurrentEvent(event));
}
```

Global variables guide the actions of HandleSDOC. The maximum number of concurrent processing threads is controlled by gMaxThreads. You can get and set this value with the convenience routines ACGIGetMaxThreads and ACGISetMaxThreads. If gRefusing is true, the handler will return the HTML page stored in gHTMLRefused and not process the event (you build this page in your custom routine WWWGetHTMLPages). You set gRefusing by calling ACGIRefuse. If you're really concerned about heap fragmentation, you might want to create a pool

of preallocated threads during initialization with the number of threads in the pool equal to gMaxThreads. Threads are recycled into the pool, limiting fragmentation. This is the approach taken by Grant Neufeld in his ACGI framework (see "Threading Apple Events" in the April 1996 issue of *MacTech Magazine*).

The globals gThreadStackSize and gThreadOptions give you control over how threads are created. The convenience routines ACGIGetThreadParams and ACGISetThreadParams allow you to get and set their values. The default stack size of 0 causes the Thread Manager to allocate a 24K stack to each thread. (Thread creation options are described in detail in "Concurrent Programming With the Thread Manager" in *develop* Issue 17.)

If your WWWProcess routine (or any routine that it calls) uses a lot of stack space for local variables, you might have to increase the thread stack size. You should do this in your WWWInit routine. You'll know if you're running out of stack space in your ACGI because your server computer will usually lock up when a running thread's stack overflows the heap space allocated to it. So remember, if your server keeps freezing up or bombing, and you don't think your code is the problem, try increasing the stack size allocated to your threads and then increase the ACGI memory allocation by roughly the increase in stack size multiplied by your chosen value of gMaxThreads.

The Thread Manager has routines that check how much stack space a given thread is using. You could therefore write a debugging macro that logs the stack space remaining before calling YieldToAnyThread. This could be useful in isolating where the problem is after the crash — but it wouldn't actually stop the thread from exhausting its stack space because that happens between yields. •

HTTP REQUEST PROCESSING

Each thread created by HandleSDOC won't start running until the main event loop calls YieldToAnyThread. When it's time for a new thread to run, the Thread Manager saves the state of the thread that just yielded, sets up the new thread's environment, and then calls SDOCThread. This routine is where all the real work of the ACGI takes place — and where your custom processing routine WWWProcess is invoked.

SDOCThread is the longest and most complicated routine in the ACGI. It's responsible for extracting all request parameters, URL-decoding the search and post arguments, packing the parameters into a WWWRequest data structure, calling WWWProcess to process the request, placing the HTML response page into the server reply, and then resuming the Apple event to send the reply back to the server.

Before looking at the code, it's a good idea to go over exactly what's packed into the 'sdoc' Apple event. A client browser asks the server to run an ACGI either by referencing the ACGI's URL or by submitting HTML form pages that specify the ACGI as its action.

A direct reference is just the URL of the ACGI:

```
http://www.test.com/test.acgi
```

To invoke an ACGI as the action of a form, you need to write HTML code like this:

```
<FORM METHOD=GET ACTION="http://www.test.com/test.acgi">
    ...form input items...
</FORM>
```

or similar code for METHOD=POST. In both cases, you can supply extra arguments to the ACGI by adding them to the end of the URL like this:

```
http://www.test.com/test.acgi$path_args?search_args
```

The *path arguments* are everything between the dollar sign ($) and the question mark (?), while the *search arguments* are everything following the question mark. The order of the $ and the ? are important. If you put the ? before the $, everything following the ? (including the $ and what comes after it) is considered part of the search arguments.

When you're using forms, you can specify a method of either GET or POST. All of your form's input variables are URL-encoded. If you specify GET, the input variables are tacked onto the end of the search arguments; if you use POST, they're placed into a separate parameter called the *post arguments* and sent separately.

URL encoding isn't particularly fancy. All it means is that the input field names and field values are written out as *name=value* pairs, and all such pairs are placed into one long parameter with each pair separated from the next by an ampersand (&). All spaces in the original input variables are replaced by plus signs (+) and any special characters are replaced by their ISO-8859 Latin-1 hexadecimal equivalents in the form %*xx* (where *xx* represents the two hex digits identifying the character).

Any or all of these arguments (if present), along with a series of parameters that describe the client browser and the server, are placed into the 'sdoc' Apple event and sent to the ACGI by the HTTP server. Each parameter is identified in the Apple event by 4-character keyword names. The ACGI passes these keyword names to the Apple Event Manager to extract the various parameters.

> **For a full description of the keywords,** refer to *Planning and Managing Web Sites on the Macintosh: The Complete Guide to WebSTAR and MacHTTP*, Chapters 13 and 15. •

The five most important keywords to be aware of are as follows:

- kPathArgsKeyword — the parameter that contains the path arguments (the text between the $ and the ?)
- kSearchArgsKeyword — the search arguments (everything after the ?)
- kPostArgsKeyword — the post arguments
- kUserAgentKeyword — the name and version of the browser that made the request
- kMethodKeyword — the name of the method (such as GET, POST, or ACTION) by which the ACGI was called

The path, search, and post arguments hold the data that makes up a request. The browser name lets you decide which HTML features you might want to include in your response page. For example, you might not want to use the latest HTML features of Netscape Navigator™ in your response page if the browser name says that the client is an old version of Mosaic that doesn't understand tables and frames.

Most of the code in SDOCThread (excerpted in Listing 4) deals with extracting parameters from the event and then breaking up the search and post arguments into *name=value* pairs.

Listing 4. The SDOCThread routine

```c
static void SDOCThread(void *threadParam)
{
  WWWRequest   request;
  Size         spaceNeeded, responseSize;
  OSErr        err;

  // [1]   Copy event and reply to local storage.
  AEParams** params = (AEParams**) threadParam;
  AppleEvent event = (*params)->event;
  AppleEvent reply = (*params)->reply;

  DisposeHandle((Handle) params);

  // [2]   Initialize request structure.
  memset(&request, 0, sizeof(request));

  // [3]   Allocate storage for params/args.
  spaceNeeded = ACGIParamSize(&event);
  request.storage = NewHandleClear(spaceNeeded);
  if (request.storage == nil) {
    char  msg[128];

    sprintf(msg, "SDOCThread: no storage memory: %lu bytes.",
        spaceNeeded);
    ACGILog(msg);
    err = ACGIReturnHandle(&reply, gHTMLNoMemory);
    gDone = true;
    goto Done;
  }
  HLockHi(request.storage);

  // [4]   Copy params/args into position.
  err = ACGICopyArgs(&event, &request);
  if (err != noErr) goto Done;

  // [5]   Decode URL-encoded search and post arguments.
  if (strlen(*request.storage + (long) request.searchArgs) > 0) {
    err = ACGIURLDecode(*request.storage + (long) request.searchArgs,
            &request.searchNum, &request.searchNames,
            &request.searchValues);
    if (err != noErr) goto Done;
  }
  if (strlen(*request.storage + (long) request.postArgs) > 0) {
    err = ACGIURLDecode(*request.storage + (long) request.postArgs,
            &request.postNum, &request.postNames,
            &request.postValues);
    if (err != noErr) goto Done;
  }
  HUnlock(request.storage);
```

(continued on next page)

Listing 4. The SDOCThread routine *(continued)*

```
// [6]   Allocate HTML response.
request.response = NewHandleClear(gHTTPHeaderLen);
if (request.response == nil) {
   gDone = true;
   err = ACGIReturnHandle(&reply, gHTMLNoMemory);
   goto Done;
}
BlockMoveData(gHTTPHeader, *request.response, gHTTPHeaderLen);

// [7]   Call the custom processor.
err = WWWProcess(&request);

// [8]   Put the response into the reply and resume the Apple event.
Done:
   if (request.storage != nil) DisposeHandle(request.storage);
   if (request.searchNames != nil) DisposeHandle(request.searchNames);
   if (request.searchValues != nil) DisposeHandle(request.searchValues);
   if (request.postNames != nil) DisposeHandle(request.postNames);
   if (request.postValues != nil) DisposeHandle(request.postValues);

   responseSize = GetHandleSize(request.response);
   if (err == noErr && request.response != nil
         && responseSize > gHTTPHeaderLen)
      err = ACGIReturnHandle(&reply, request.response);
   else
      switch (err) {
         case errWWWNoMemory:
            err = ACGIReturnHandle(&reply, gHTMLNoMemory);
            break;
         case errWWWRefused:
            err = ACGIReturnHandle(&reply, gHTMLRefused);
            break;
         case errWWWTooBusy:
            err = ACGIReturnHandle(&reply, gHTMLTooBusy);
            break;
         case errWWWUnexpected:
            err = ACGIReturnHandle(&reply, gHTMLUnexpectedError);
            break;
         default:
            err = ACGIReturnHandle(&reply, gHTMLUnexpectedError);
            break;
      }
   if (request.response != nil) DisposeHandle(request.response);

// [9]   Put error code into the Apple event (if needed).
if (err != noErr) {
   long errorResult = err;    // Must be long integer.
   AEPutParamPtr(&reply, keyErrorNumber, typeLongInteger,
      &errorResult, sizeof(long));
}
```

Listing 4. The SDOCThread routine *(continued)*

```
// [10]  Resume the event, decrement running thread count, write to
//       the log.
AEResumeTheCurrentEvent(&event, &reply,
                        (AEEventHandlerUPP) kAENoDispatch, 0);
gThreads--;
ACGILog("Done.");
return;
}
```

The only item passed to your custom WWWProcess routine is a pointer to the WWWRequestRecord. You access the items stored in the record using the convenience routines that are defined later.

EXTRACTING PARAMETERS FROM THE APPLE EVENT

The routines ACGIParamSize and ACGICopyArgs repeatedly call the Apple Event Manager to get the size and the text of each parameter. ACGICopyArgs moves each successive parameter into the **request.storage** handle in the WWWRequestRecord data structure (see acgi.h). It also places the offset of each parameter, relative to the start of the handle, into corresponding pointer variables in **request**. Because most parameters are only 10 to 100 bytes in length, it seemed far more efficient to pack them all into a single handle. This avoids the overhead of making multiple calls to the Memory Manager to allocate one handle for each parameter and then make multiple calls to HLock and HUnlock when manipulating the parameters during processing.

> **Another way of storing the parameters** is to place each parameter into its own handle. See Jon Norstad's Mail Tools code (available at http://charlotte.acns.nwu.edu/mailtools/techinfo.html) for an example of this other approach. •

All parameters are stored as text strings, even the connection ID (a long integer). Missing or empty parameters are stored as zero-length strings so that the ACGI can handle requests from HTTP servers that only partially implement the full WWWΩ Apple event suite (there's no guarantee a given server program will pass your ACGI all the parameters defined in the suite). You can get the numeric value of any parameter by calling the convenience routine HTTPGetLong.

DECODING URL-ENCODED POST ARGUMENTS

The search and the post argument strings are URL-decoded by the routine ACGIURLDecode following the prescription outlined in Chapter 13 of *Planning and Managing Web Sites on the Macintosh: The Complete Guide to WebSTAR and MacHTTP*.

The routine begins by counting all of the *name=value* pairs in the given string by looking for & separators. Two handles are then allocated to hold the **char*** pointers. The string is then scanned, and the offset of each argument name and its associated value are recorded in the arrays. Finally, the routine ACGIDecodeCStr is called to convert each *name=value* pair from ISO-8859 Latin-1 encoding to the standard Macintosh Roman encoding. The conversion table used by the popular Netscape Navigator browser is employed here for compatibility. If you want to substitute another 256-character translation table, you'll need to replace the ID=1000 'xlat' resource located in the resource file acgi.rsrc.

CONVENIENCE ROUTINES

There are three sets of convenience routines that allow you to extract parameters from a server request, build your HTML response page, and fine-tune the runtime performance of the ACGI.

PARAMETER AND ARGUMENT EXTRACTION ROUTINES

Seven routines, identified by the prefix "HTTP," can be used to extract parameters or post and search arguments from the WWWRequestRecord that's passed to the WWWProcess routine. The enumeration WWWParameter contains the name by which an individual parameter must be referenced:

```
typedef enum WWWParameter {
    p_path_args = 0,
    p_username,
    p_password,
    p_from_user,
    p_client_address,
    p_server_name,
    p_server_port,
    p_script_name,
    p_content_type,
    p_referer,
    p_user_agent,
    p_action,
    p_action_path,
    p_method,
    p_client_ip,
    p_full_request,
    p_connection_id
} WWWParameter;
```

Following are descriptions of the routines.

```
Boolean HTTPLockParams(WWWRequest r);
```

Locks down the request parameters. Several items in the WWWRequestRecord are stored as handles and must be locked down before the ACGI can access them. HTTPLockParams locks the items down for you and HTTPUnlockParams (below) releases them. It might be a good idea to unlock your parameters before calling YieldToAnyThread.

Convenience routines that return **const char*** pointers to parameters implicitly call HTTPLockParams to lock down the WWWRequestRecord before they return the pointers. Note that the request record remains locked when the routines return. The routines that copy parameters and arguments into the character strings you pass in will lock the request record while they're copying the information and then unlock it before they return (but only if the data structure wasn't already locked on entry).

```
void HTTPUnlockParams(WWWRequest r);
```

Unlocks the request parameters.

```
const char *HTTPGetParam(WWWRequest r, WWWParameter par);
```

Gets a pointer to one of the parameter strings. This leaves **r** locked.

```
Boolean HTTPGetLong(WWWRequest r, WWWParameter par, long *i);
```

Gets the integer value of a parameter. The result is returned in **i**. The routine returns false if the parameter is not an integer.

```
Boolean HTTPCopyParam(WWWRequest r, WWWParameter par, char *result, long len,
    long *actualLen);
```

Copies the parameter text into the character variable **result**. The length of **result** is in **len**; the actual length of the parameter is returned in **actualLen**. The routine returns false if the parameter identifier **par** is invalid.

```
long HTTPGetNumSrchArgs(WWWRequest r);
long HTTPGetNumPostArgs(WWWRequest r);
```

Gets the number of search or post arguments.

```
Boolean HTTPGetSrchArgAt(WWWRequest r, long index, char *name, long nameLen,
    long *actualNameLen, char *value, long valueLen, long *actualValueLen);
Boolean HTTPGetPostArgAt(WWWRequest r, long index, char *name, long nameLen,
    long *actualNameLen, char *value, long valueLen, long *actualValueLen);
```

Gets a search or post argument by absolute position. **index** is between 1 and the total number of such arguments. **name** receives the name of the argument at position **index**, and **value** receives the value. The lengths of the character array's name and value are in **nameLen** and **valueLen**. The actual lengths of the items are returned in **actualNameLen** and **actualValueLen**. The routine returns false if **index** is out of range.

```
Boolean HTTPGetSrchArgCount(WWWRequest r, char *name, long *numValues);
Boolean HTTPGetPostArgCount(WWWRequest r, char *name, long *numValues);
```

Gets the number of search or post arguments that have the field name **name**. The number is returned in **numValues**. The routine returns false if there's no search or post argument called **name**.

```
const char *HTTPGetMultipleSrchArg(WWWRequest r, char *name, long index);
const char *HTTPGetMultiplePostArg(WWWRequest r, char *name, long index);
```

Tries to get the instance **index** of a multivalued search or post argument. The routine returns an empty string if **index** is out of range or if **name** doesn't exist. The routine leaves **r** locked on exit. **index** starts at 1.

```
Boolean HTTPGetLongMultipleSrchArg(WWWRequest r, char *name, long index,
    long *i);
Boolean HTTPGetLongMultiplePostArg(WWWRequest r, char *name, long index,
    long *i);
```

Gets the integer value of the instance **index** of a multivalued search or post argument called **name**. The routine returns the value in **i**, and returns false if **index** is out of range or the argument is not an integer. **index** starts at 1.

```
Boolean HTTPCopyMultipleSrchArg(WWWRequest r, char *name, long index,
    char *value, long len, long *actualLen);
Boolean HTTPCopyMultiplePostArg(WWWRequest r, char *name, long index,
    char *value, long len, long *actualLen);
```

Copies the contents of the instance **index** of a multivalued search or post argument called **name**. The routine returns text in **value**. The length of the **value** string is in **len**; the actual length of the **value** string is returned in **actualLen**. The routine returns false if **index** is out of range or **name** doesn't exist. **index** starts at 1.

HTML PAGE COMPOSITION ROUTINES

There are ten routines, all prefixed with "HTML," to help you compose the HTML response pages. The routines that allow you to append different types of data to the response page are shown in Table 1; the handle to the response page is obtained by calling HTMLGetResponseHandle.

```
Handle HTMLGetResponseHandle(WWWRequest r);
```

Gets the handle to the HTML response page.

```
OSErr HTMLClearPage(Handle r);
```

Clears the current response page (except for the HTTP header) and starts over.

Table 1. Routines that append data to the HTML response page

Routine	Appends to response page
OSErr HTMLAppendHandle(Handle r, Handle h);	Contents of handle **h**
OSErr HTMLAppendTEXT(Handle r, long iTEXTResID);	TEXT resource with ID iTEXTResID
OSErr HTMLAppendString(Handle r, long iSTRResID);	STR resource with ID iSTRResID
OSErr HTMLAppendIndString(Handle r, long iSTRResID, long index);	String at location **index** in STR# resource with ID iSTRResID
OSErr HTMLAppendFile(Handle r, char *localFileName);	Local text file
OSErr HTMLAppendCString(Handle r, char *cString);	C string
OSErr HTMLAppendPString(Handle r, StringPtr pString);	Pascal string
OSErr HTMLAppendBuffer(Handle r, char *buffer, long len);	Text buffer of length **len**

ACGI RUNTIME-TUNING ROUTINES

There are 13 routines that allow you to fine-tune the runtime behavior of the ACGI without having to modify the code in acgi.c or directly set global variables.

```
void ACGIShutdown(void)
```

Shuts down the ACGI as soon as all current threads are finished.

```
Boolean ACGIIsShuttingDown(void)
```

Tests whether the ACGI is shutting down.

```
Boolean ACGIRefuse(Boolean refuse)
```

Sets whether to accept or reject requests.

```
unsigned long ACGIGetRunningThreads(void)
```

Gets the number of active threads.

```
unsigned long ACGIGetMaxThreads(void)
void ACGISetMaxThreads(unsigned long newThreads)
```

Gets or sets the maximum number of threads allowed to run at the same time.

```
void ACGIGetSleeps(long *whenThreads, long *whenIdle)
void ACGISetSleeps(long whenThreads, long whenIdle)
```

Gets or sets the sleep settings.

```
long ACGIGetWNEDelta(void)
void ACGISetWNEDelta(long newDelta)
```

Gets or sets the time between calls to WaitNextEvent.

```
void ACGIGetThreadParams(Size *stack, ThreadOptions *options);
void ACGISetThreadParams(Size stack, ThreadOptions options);
```

Gets or sets the thread stack size and creation options.

```
const char *ACGIGetHTTPHeader(void)
```

Gets a pointer to the standard HTTP header.

CUSTOMIZABLE ROUTINES

The six customizable routines in www.c allow you to adapt the ACGI shell to suit your needs. I've supplied simple, straightforward samples of the routines in the file www.c.

The default version of the WWWProcess routine is shown in Listing 5. It returns a page that displays all of the HTTP server request parameters in a nicely formatted table. Note the use of the YIELD macro here. It provides a convenient way of yielding to other threads and automatically aborting should the ACGI signal that it wants to quit.

Listing 5. The default version of WWWProcess

```
#define YIELD() { YieldToAnyThread(); \
                  if (ACGIIsShuttingdown()) \
                      return (errWWWRefused); }

OSErr WWWProcess(WWWRequest request)
{
    Handle   r = HTMLGetResponseHandle(request);
    char     s[1024], name[512], value[512];
    long     len, i, n, iName, iValue;
    Boolean  gotOne;
    OSErr    err;

    // Build a table to display the WebSTAR request parameters.
    err = HTMLAppendPString(r,
        "\p<HTML><HEAD><TITLE>ACGI</TITLE></HEAD>\r\n");
```

Listing 5. The default version of WWWProcess (continued)

```
        YIELD();
        err = HTMLAppendCString(r,
          "<BODY><H1>ACGI Parameters</H1><TABLE BORDER=0>");
        YIELD();
        err = HTMLAppendCString(r,
          "<TR><TD ALIGN=RIGHT NOWRAP><B>Path arguments:</B></TD><TD>");
        YIELD();

        if (HTTPCopyParam(request, p_path_args, s, 1023, &len))
          err = HTMLAppendCString(r, s);
        YIELD();

        ...   // and so on, for all the other parameters

        // Now show all the search arguments.
        err = HTMLAppendCString(r,
          "</TD></TR><TR><TD ALIGN=RIGHT NOWRAP VALIGN=TOP>"
          "<B>Search Arguments:</B></TD><TD>");
        YIELD();

        n = HTTPGetNumSrchArgs(request);
        if (n > 0) {
          for (i = 1; i <= n; i++) {
            gotOne = HTTPGetSrchArgAt(request, i, name, 511, &iName, value,
                                     511, &iValue);
            if (gotOne) {
              if (i > 1)
                err = HTMLAppendCString(r, "<BR>");
              err = HTMLAppendCString(r, name);
              err = HTMLAppendCString(r, " = ");
              err = HTMLAppendCString(r, value);
            }
            YIELD();
          }
        }
        else
          err = HTMLAppendCString(r, "(none)");

        ...   // and similarly for the post arguments

        err = HTMLAppendCString(r,
          "</UL></TD></TR></TABLE>\r\n</BODY>\r\n</HTML>\r\n");
        return (err);
}
```

OVER TO YOU

That's about it for writing threaded, high-performance ACGIs in C. I bet you thought it was a lot more difficult than this, didn't you?

A threaded ACGI written in a high-level language offers a significant performance increase compared to an equivalent ACGI written in AppleScript. If you've been

using AppleScript exclusively to do your HTML form processing, I hope this article will whet your appetite to try something a bit more daring. It's time to kick your Web site into high gear and move it over into the fast lane!

REFERENCES

- "Futures: Don't Wait Forever" by Greg Anderson, *develop* Issue 22.

- "Concurrent Programming With the Thread Manager" by Eric Anderson and Brad Post, *develop* Issue 17.

- *MacTech Magazine* articles: "Threading Apple Events" by Grant Neufeld, Vol. 12, No. 4 (April 1996); "Writing CGI Applications in C" by John O'Fallon, Vol. 11, No. 9 (September 1995); and "Adding Threads to Sprocket" by Steve Sisak, Vol. 10, No. 12 (December 1994). *MacTech Magazine* articles can be found at http://web.xplain.com/mactech.com/magazine/features/articlearchives.html.

- *Planning and Managing Web Sites on the Macintosh: The Complete Guide to WebSTAR and MacHTTP* by Jon Wiederspan and Chuck Shotton (Addison-Wesley, 1995). Chapter 13, "Writing CGI Applications," and Chapter 15, "Developing CGIs in C," are available in the file "Developing CGIs.pdf" included in the WebSTAR documentation at ftp://ftp.starnine.com/pub/docs/webstar_doc.sea.hqx.

ACCORDING TO SCRIPT

User Interactions in Apple Event– Driven Applications

CAL SIMONE

So far throughout the history of the Macintosh, most applications have been designed to be run by a user double-clicking the icon of an application (or document) in the Finder and manipulating the application and its data through the graphical interface. Recently, the publishing industry has adopted AppleScript and scriptable applications as the mechanism for creating production systems. Now that scripting and Apple events have become more pervasive, and more and more applications are scriptable, applications must be prepared to be controlled remotely by Apple events from other applications and scripts. And there's a new kind of application on the horizon, the Apple event–based *server application*. A server application has no user interface: it's designed to communicate with the outside world only through Apple events. Although server applications have been possible to write since the introduction of System 7, they're becoming increasingly important and will play a major role in the future versions of the Mac OS.

In other words, there may not be a human being sitting at the computer where your application is running.

In this column, I'll cover these topics:

- the two types of Apple-event control

- determining when your application should interact with a user

- how to interact, when a user is present

- how *not* to interact, when no user is present, and how to return errors when you don't interact

There are some important differences between direct manipulation through the graphical interface and remote control through interapplication communication. When developing server applications, you'll want to be careful about what happens when an Apple event (or series of Apple events) takes over. This applies not only to applications that are designed to be controlled primarily through Apple events, but also to those designed to be controlled mainly through a graphical interface but for which Apple events provide an alternative interface to the graphical one. The information presented in this column applies to both types of application.

TYPES OF APPLE-EVENT CONTROL

For this discussion, I'll classify Apple-event control into two scenarios:

- Simulating a user sitting at the computer — Many users will write scripts that help them automate their work. They'll generally still be at (or near) the computer when the scripts run. In such cases, it's OK (or even expected) that an application will interact with the user whenever there's a problem or a choice needs to be made, or when the computer needs more input. Many scripts that drive such applications will even incorporate interaction in the form of dialog boxes. This scenario is often the case for embedded scripts, such as those attached to tool palette icons.

- When a user isn't present — This is the more interesting case for automation and integration through interapplication communication, and will become increasingly common as more intelligent scripts are written. Some operations (and some applications) are more suited than others for unattended batch processing. It's recommended for operations that take a long time, such as those found in production systems; in these situations, direct interaction with the user is usually not a good idea, since no one's likely to be there to deal with a problem or a request for more information. Server applications deal only with this scenario.

Since your application may need to respond to either type of Apple-event control, I'll describe how to comfortably handle both, starting with the way to determine which one you're dealing with at a particular moment. The largest issue to tackle in responding to Apple events is whether to interact with the user. I'll explain how to determine whether you should interact and what to do once you've made that determination. Incidentally, in a standard factored application (you *are* factoring your application these days, aren't you?) you can get most of the proper behavior for free.

CAL SIMONE (mainevent@his.com) eats, drinks, and sleeps AppleScript. Just when he thought he was out of the woods, the AppleScript Language Association (ASLA) was born. Oh well, maybe next year he'll get to take that vacation. And why didn't Cal write a column for the last couple of issues of *develop*? Let's see... his company shipped a product, and he participated in six trade shows, moved to a new apartment, and solved the world hunger problem (just kidding about that last one). •

An example scenario. Let's look at a common case that may or may not require user interaction: handling the Core suite's Close event. According to the *Apple Event Registry*, one of the optional parameters for this event is the **saving** parameter, which can have one of three enumerated values: **yes**, **no**, and **ask**. The traditional meanings of these values are as follows:

- **yes** will always save a modified document, presenting a standard file dialog if it has never been saved (unless the user included the **saving in** parameter to specify where to save the document).

- **no** will never save a modified document, but rather discard any changes.

- **ask** will allow the user to choose whether to save the document, invoking the user interface behavior.

But it's not quite that simple. For example, assume that the user modifies a document in one of your application's windows and then runs a script that executes the command **close the front window** (with no **saving** parameter or with **saving ask**). Your application's Close event handler will typically display a dialog box asking the user whether to save the document. The catch is that you can't always do this; it depends on where the script is running. You could implement a preference setting that allows the user to configure your application for "response to humans vs. non-humans," but having too many preferences spoils the simplicity.

So what should you do? The solution is to call the routine AEInteractWithUser, and then decide what to do based on its return result. But before getting into that, we'll take a look at how the Apple Event Manager decides whether user interaction is appropriate.

BEHIND THE SCENES
The following conditions are considered by the Apple Event Manager to determine whether you should interact while handling an Apple event. Note that the first one is an application setting, while the last two are optional attributes of a particular event that may be set by the sender.

- the user interaction level — whether your application allows interactions in general

- the event source attribute — where the particular Apple event came from

- the event's interaction-requested attribute — whether the Apple event wants you to interact

The user interaction level. The Apple Event Manager first checks the *user interaction level* of your application. You can call AEGetInteractionAllowed yourself to determine which Apple event sources can cause your

application to interact with the user. If you need to change the interaction level, you can set it at any time with AESetInteractionAllowed.

The Apple Event Manager provides a data type, AEInteractAllowed, which is an enumeration that defines three levels of allowable interaction:

- Interact with self — At this level, you can interact with the user only in response to Apple events you've sent to yourself, through your factored user interface or from an attached or embedded script that you're executing inside your application.

- Interact with local — You can interact with the user in response to Apple events originating from the same computer where your application is running. This includes the above case.

- Interact with all — You can interact with the user in response to any Apple event, whether sent from the same computer or a remote machine.

Remote events will arrive only if all the conditions for accepting remote events are met. You need to set two flags in the SIZE resource — "accept high-level events" (to receive *any* Apple events) and "allow local and remote events" (to receive events from other computers) — and give permission for program linking in the Users and Groups control panel. You must also make sure that Program Linking is turned on in the Sharing Setup control panel and enabled for the application in the Finder's Sharing dialog. •

There's a fourth possibility for some applications, the true lowest level of interaction, which is "no interaction." This is the appropriate level in a background-only application or any pure server application, or any other situation where interaction is undesirable. Consider this example (which Jon Pugh came across when he was working for Storm Technologies, while implementing scriptability in PhotoFlash): An attached script is executed as part of an automation process. The script gets an error and the application puts up a dialog box — but there's no one there to answer it. If users can execute a script inside your application and the script might be run without a user present, you'll have this problem. Before running the script, the user needs to be able to tell your application, "No one will be here, so don't interact."

Since the Apple Event Manager doesn't provide support for the no-interaction condition (the AEInteractAllowed type has only three possible values), you'll have to set up and maintain this yourself. The simplest way to implement this setting is by using a global Boolean variable for the no-interaction flag. If the variable's value is true and your application is called on to do interaction, you'll know right away not to allow the

interaction. (Note that if your application handles multiple concurrent Apple events using threading, an application global is *not* a good solution.)

Because AppleScript doesn't provide a way to get or set interaction levels from scripts, you'll need to implement a **user interaction level** property for your application, similar to the one in PhotoFlash, that a user can set or get from a script. This property should handle all four enumeration constants and associate the fourth level, no interaction, with the global Boolean variable. In your 'aete' resource, use the following terminology and 4-byte codes for the enumeration constants:

- **never interact** — 'eNvr'
- **interact with self** — 'eInS'
- **interact with local** — 'eInL'
- **interact with all** — 'eInA'

Apple events initiated from the user interface should probably ignore the no-interaction flag altogether, and just call AEInteractWithUser and interact in the usual way if need be. This way your application could perform double duty, successfully performing actions initiated by Apple events from other sources without disturbing a user who might be sitting there using the application.

The event source attribute. The next step taken by the Apple Event Manager is to examine the *event source attribute* of the particular Apple event being handled. If you want to look at the source for an Apple event, you can call AEGetParamPtr with the keyEventSourceAttr keyword to obtain the source. The source data type, AEEventSource, is an enumeration indicating five possible sources: unknown, direct call, same process, local process, and remote process. This is checked against the user interaction level to find out whether your application allows user interaction in response to an Apple event from this particular source.

The interaction-requested attribute. Finally, if the Apple Event Manager determines that interaction with the user in response to the event source is OK, it examines the event's *interaction-requested attribute*, which tells the Apple Event Manager what kinds of interaction are requested by the Apple event. If you want to look at this level yourself, you can call AEGetParamPtr with the keyInteractLevelAttr keyword to obtain the interaction level requested by the Apple event. There are three constants that represent the interaction levels:

- kAEAlwaysInteract — Your application can interact with the user for any reason, such as to confirm an action, notify the user of something, or request information.

- kAECanInteract — Your application can interact with the user if it needs to request information from the user to continue.

- kAENeverInteract — Your application should never present a user interface while handling this Apple event.

When present, an additional constant that's set by the sender of the event, kAECanSwitchLayer, contributes to determining whether you may bring yourself to the foreground if you need to interact.

If the interaction-requested attribute is present, both its value and the user interaction level (obtained from AEGetInteractionAllowed) determine whether you can interact. Otherwise, the default rules apply: if the event is remote, the default is "never interact"; otherwise the default is "can interact." This default scheme has the advantage that the interaction levels work out neatly for events sent to your application from yourself — that is, from attached or embedded scripts running inside your application or from your factored user interface — since the default for events on the same machine is "can interact." (Note that the "never interact" value for the interaction-requested attribute is different from the "no interaction" user interaction level described earlier. The former is an event attribute set by the sender of the event, while the latter is a setting in the server application.)

DETERMINING WHETHER TO INTERACT
So how do you make this work in your application? Before initiating user interaction, you'll need to make sure your application is the active application — that is, it's in the foreground. Let's take a moment to discuss the ways to become the active application.

In the old days, shortly after System 7 was released, developers used to call the GetCurrentProcess routine, followed by SetFrontProcess to switch layers. In that case, an application won't actually go to the foreground until the next call to WaitNextEvent, which usually won't happen until the application returns to the main event loop. Moreover, you can't later prevent your application from coming to the foreground if you're able to correct a problem and no longer need to be in the foreground. Since we're talking here about the situation where you're in an Apple event handler when this happens, *don't* call SetFrontProcess to make yourself the active application; there's no need to do this. Developers have also used the Notification Manager to put up an alert, beep, place a diamond mark in the Application menu, call Mom in Florida, and so on. This is somewhat better, but still more difficult than it has to be.

It turns out that finding out whether you should interact with the user and getting yourself into the foreground are really easy with the Apple Event Manager. What? "Apple Event Manager" and "easy" in the same sentence? Yes, it's true! The correct, easy, and fun way to do this is by calling AEInteractWithUser (which will call the Notification Manager on your behalf to get the user's attention):

```
err := AEInteractWithUser(timeOutInTicks,
   notificationRec, idleProc);
```

What's more, this works even in situations where there's no Apple event being handled (whenever you need to gratuitously get yourself into the foreground, right where you are in your code).

If AEInteractWithUser returns noErr, you're in the foreground and it's safe to interact. That's it! The conditions discussed earlier are automatically tested for you, so you won't have to do any of that yourself.

Be sure to check the error that AEInteractWithUser returns. If it's not appropriate for you to interact, AEInteractWithUser returns errAENoUserInteraction. Also, be aware that AEInteractWithUser can time out if timeOutInTicks is reached before the user responds to the notification. You should supply a reasonable timeout value for timeOutInTicks, and if AEInteractWithUser times out, return errAETimeout or some other suitable error as the result for your event handler. And remember

that the original Apple event can time out if the notification is outstanding for too long.

Listing 1 shows a very simple routine that calls AEInteractWithUser, checking the no-interaction flag if requested. You should call this routine before every alert or dialog box that you may present, and handle the situation another way if you aren't allowed to interact.

So, in our earlier Close event example, the **saving yes** and **saving ask** cases work out a little differently than you might have expected: if AEInteractWithUser returns noErr, put up the alert or dialog; if not, return from the Close event handler with errAENoUserInteraction, errAETimeout, or some other suitable error.

WHEN THE USER IS THERE
Now that you've determined whether or not to interact in a given situation, let's look at a couple of things to keep in mind when you put up an alert or dialog box while handling an Apple event.

Listing 1. Interacting with a user while handling an Apple event

```
FUNCTION AllowInteraction(timeOutInTicks: LongInt; checkNoInteractFlag: Boolean): Boolean;
VAR
   procSerNum: ProcessSerialNumber;

BEGIN
   (* If the no-interaction flag should be checked and that flag is true, don't allow any
      interaction. *)
   IF checkNoInteractFlag & gNoInteract THEN
      AllowInteraction := false
   ELSE
      BEGIN
         (* After executing the next line, your application should be in the foreground, unless it
            times out, or unless interaction isn't appropriate. Note that gIdleProc is usually your
            Apple event idle proc. *)
         err := AEInteractWithUser(timeOutInTicks, gNotificationRec, gIdleProc);
         IF err <> noErr THEN
            ...   (* errAETimeout, or some other error *)
         AllowInteraction := err = noErr;
      END;
END;
```

If you end up presenting an alert or dialog box in response to an Apple event for which you shouldn't interact (because for some reason you weren't sure whether you should interact or not), it's best to time yourself out by dismissing the dialog after a reasonable time, even when no one has clicked any of its buttons. You might consider doing this even if interaction is allowed; after a lengthy time, put your application out of its misery and move on. Just be careful which button you choose — you might want the effect of canceling out the dialog, or producing the least damage, or losing the least amount of work. If you choose to continue processing, be prepared to use suitable defaults, if applicable.

An example of an Apple event scenario that *always* involves interaction with the user is handling the Edit Graphic Object (EGO) event. EGO is an Apple event specification devised by Allan Bonadio in 1991. In the EGO scenario, an application that contains an embedded object that was created by another application allows a user to edit the object in the creating application. If you do need to pull yourself to the front immediately (such as with EGO), bypassing any possibility of Notification requests, you should call SetFrontProcess followed by AEInteractWithUser. You can insert the following just before the call to AEInteractWithUser in Listing 1 (in C, the parameter procSerNum in the GetCurrentProcess call needs to be a pointer):

```
err := GetCurrentProcess(procSerNum);
err := SetFrontProcess(procSerNum);
```

Because the Apple event always results from a user action, in this situation it's OK to force yourself into the foreground — this may be the only reasonably significant case where you should do this.

And, as in the case of AEInteractWithUser, you should be careful about pending update events while your alert or dialog box is on the screen.

WHEN NOBODY'S HOME

What if there isn't a user sitting at the computer? You'll want your application to cope with limitations or restrictions on user interaction without failing, especially when it comes to reporting errors properly.

When your application is more likely to be controlled by a batch process, such as a script that runs periodically in a production system, avoid requesting user interaction when the computer is likely to be unattended. (When you're not sure how you're being controlled, performing the tests discussed earlier in "Behind the Scenes" may help you find out.) You don't want an alert popping up on a screen when there's no one to respond — this can cause the computer to come to its knees and, depending

on what other applications are running, may even cause it to stop processing anything else.

Dealing with the problem yourself. Not being able to interact when you want to is a problem that can arise from any of the following: you're faced with the error errAENoUserInteraction (or your AllowInteraction function returns false); AEInteractWithUser times out; or you implement "no interaction" as a user interaction level. In some cases, the best thing to do is to try to handle the situation yourself.

If the situation doesn't require the user to make a choice or supply any information — that is, if you just want to tell the user something — don't hold up the works for this; simply skip the interaction. But if you do require a choice or some information, consider handling this situation intelligently in the application. Users will have a better experience if they come back from lunch (or the next day) to find that your application carried on instead of stopping after the first 20 seconds because it needed some small piece of information. For example, if an Apple event is missing a required parameter, and the inability to put up a dialog box means that you can't request information from the user, see whether you can use default values instead.

Be careful what behavior you choose when the user isn't available to decide. And don't go overboard by trying to second-guess a user — that can cause genuine irritation. If you really, really need to interact but you can't, it's acceptable to generate an error such as errAENoUserInteraction. Judge what's best for your application, based on who uses it and how it's used.

Returning an error to the Apple event. Sometimes the best way to avoid user interaction is to return an error describing what went wrong through your Apple event reply. For server applications, this is the *only* way to interact. It puts the burden of responsibility in the hands of the Apple event's sender, which is often a script or another application running on another machine. This way, you're off the hook — the decision about whether to find and disturb a user is made by another process. As shown in Table 1, the Apple event error-reporting mechanism provides several *error parameters* that you can use to more concisely describe the nature of the problem.

It's important to come up with unique error numbers, because the text for the corresponding error messages should be localized and thus may vary from location to location and system to system. The numbers are the best means for the sending application or script to trap errors, since they won't vary.

Table 1. Apple event error parameters

Parameter	AppleScript keyword	Meaning
keyErrorString or kOSAErrorMessage	direct parameter	The error message text. Strive to keep this clear and concise.
keyErrorNumber or kOSAErrorNumber	number	The error number. Remember that Apple reserves all negative values, as well as positive values up to 128.
kOSAErrorPartialResult	partial result	If more than one item is being handled by the Apple event, you can return the successful results processed so far.
kOSAErrorOffendingObject	from	You can indicate which object has caused the problem. This is especially useful for coercions.
kOSAErrorExpectedType	to	In coercions, the type requested in the coercions.

Note: The AppleScript keywords are those used in the **error** and **on error** commands.

Normally the error code you return from your Apple event handler becomes the error number, so there's no need to explicitly supply an error number parameter. It's better to use the error number as the return value from your event handler, rather than stuffing it as a parameter in the reply event yourself, because the error code will become the return value for the call to AESend in the sending application or script, which can immediately detect that something went wrong with the handling of the Apple event. (However, if you suspend an Apple event, you must place the error number explicitly in the error number parameter of the reply before resuming the event.) In any case, all other error parameters must be placed in the reply event with AEPutParamPtr or AEPutParamDesc. As an example of setting an error parameter, here's how to provide an error string to an Apple event reply:

```
err := AEPutParamPtr(reply, keyErrorString,
   typeChar, @messageStr[1], length(messageStr));
```

If you want to cancel an operation, you'll typically return error -128 (userCanceledErr), which is the most reliable way to stop executing a script that's sending an Apple event to your application. The exception to this rule occurs when the Apple event was sent from a script command that's inside a **try** block that specifically traps for that error; in this case, you may not be able to halt the script.

Note that there are many cases where either you can't return errors or, if you can, they'll be ignored. An Apple event sent with a send mode of kAENoReply doesn't have a reply event. So if you try to stuff parameters such as error parameters into this nil reply, you'll die horribly. You can also get a nil reply event if the Apple event is sent by an AppleScript command in an **ignoring application responses** block. Furthermore, the Finder ignores errors returned to Apple events it sends, such as the Required suite's Open Documents event that's sent when icons are dragged onto your application's icon. And a user can trip you up by enclosing an AppleScript command in a **try** block (**try…on error…end try**); you'll still get a reply event, but the script that executes the command can ignore any error information you return in the reply, or ignore your error altogether. There's no specific strategy to follow in these cases, since you can't know when this is happening, but be aware that it can happen.

Supplying a missing data value. Part of the design of your error-handling scheme includes determining that it isn't always necessary to return an error. Rather than cause an error, sometimes it's better to return noErr and place a Boolean indicator value or an empty list in the direct parameter as the result. This depends mostly on which Apple event you're handling.

For example, if you receive a request to find or use an object or a file that doesn't exist, you should return an appropriate error, such as errAENoSuchObject or fnfErr. (The exception to this is the Does Object Exist event, where you always want to return a true or false value.) On the other hand, if the request is to search for all items matching a certain criterion, you may want to return an empty list if nothing matches the criterion. This way the sending application or script won't fail, and the script can just check the result instead of having to trap for errors.

OUT ON GOOD BEHAVIOR

When a user is likely to be at the computer while a script controlling your application is run or while another application is directly communicating with your application, interacting with the user can be appropriate. Although having the application bring up

alerts and dialogs used to be the norm, it's becoming rarer. Since scripts can interact with a user, why not let the script do the user interaction when it needs to?

The techniques I've described in this column apply whether Apple events are an alternative to the graphical interface or your application is designed specifically as an Apple event server application. If your application will be controlled through Apple events, decide when it's appropriate (and when it's not appropriate) to interact with the user, by performing the tests discussed in this column. A well-planned interaction and error code-and-message scheme can make it possible for users to run your application in situations where a user isn't in control.

Thanks to Andy Bachorski, Sue Dumont, and Jon Pugh for reviewing this column. •

For recently updated vocabulary advice from Cal, see this issue's CD or *develop*'s Web site, under Additional Articles. •

Using Newton Internet Enabler to Create a Web Server

With Newton Internet Enabler (NIE), a whole host of TCP/IP-based applications become possible on the Newton. Internet clients for e-mail, Web browsers, and other common services are springing up, and developers are using NIE to provide portable access to legacy systems and databases. But how do you get started? This article will explore the details of using NIE by presenting a sample Internet application — a working Web server.
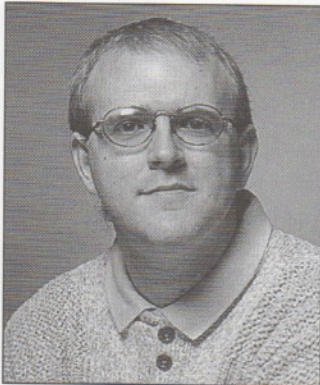
RAY RISCHPATER

The long-awaited Newton Internet Enabler (NIE for short) opens up the Internet to Newton users — the personal communications platform can now speak TCP/IP to the rest of the world. With so many possible applications that could use it, programming under NIE is not only useful, it's sure to be a lot of fun!

I've chosen to introduce NIE by looking at a sample application called nHTTPd — a Newton-based Web server that implements the HTTP protocol. Throughout the article, I'll assume you're familiar with the basics of Newton software development and TCP/IP; see the *Newton Programmer's Guide* and the NIE documentation for reference.

SAMPLE APPLICATION BASICS

Before delving into the sample application, let's take a moment to see what's included in NIE. NIE provides support for three distinct entities. There's a Link Controller for sharing an underlying link (that is, the low-level PPP or SLIP connection) between multiple applications and performing expect-response scripting. It also provides a Domain Name Service (DNS) to map back and forth between host names and IP addresses. Finally, of course, there's an implementation of TCP and UDP through NewtonScript's endpoints. For details of how the different parts of NIE work, see "NIE in a Nutshell."

Our sample application (which accompanies this article on this issue's CD and *develop*'s Web site) is a bare-bones HTTP version 0.9 server. To help you understand the code, here are the basics of the HTTP 0.9 protocol:

- Objects (usually Web pages) are referred to by a URL, which is a concatenation of the protocol being used (http), the host name serving the document (such as www.lothlorien.com), and the path and filename of the document

RAY RISCHPATER (ray_rischpater@allpen.com) is a Software Craftsman currently employed at AllPen Software, Inc., developing custom applications for the Newton platform. His hobbies include writing freeware for PDA devices, amateur radio, and channeling technical articles for journals such as *develop* and *PDA Developers* from his Siberian Husky, Sake.•

being served (such as "/" for the root document). A typical URL is http://www.lothlorien.com/.

- Objects are fetched with the GET instruction. A request comes in containing the word GET followed by the partial URL (the path and filename, but not the protocol or host name) of the object being fetched, followed by a carriage return and linefeed pair. For example:

```
GET /
```

- Form data can be retrieved with an extension of the URL: the URL is followed by a question mark and some text. Within this text, field names may be declared to the left of an equal sign (=), with the data in the field to the right of the equal sign, although data is not required. Multiple field declarations are separated by an ampersand (&). A form with two fields might look like this:

```
GET /myapp/search.html?field1=hello&field2=world
```

In response, the HTTP server dynamically creates a response object that's returned to the client.

For the latest version of the sample application, check out the latest CD and *develop*'s Web site, as this code is perpetually evolving.•

HANDLING REQUESTS

Since the Newton doesn't have conventional directories or files, we need to make some compromises to translate directory- and file-oriented specifications into something more meaningful on a Newton. My implementation uses a registry to correlate Newton data (such as application soup entries) with translators capable of creating Web objects from Newton data. These translators convert frames or soup entries to HTML or pure text, and nHTTPd has an API with three methods to allow other programs to register as translators:

- nHTTPd:RegTranslator(inAppSym, inPathStr, inCallbackSpec) registers the callback specification frame inCallbackSpec for your application (whose application symbol is inAppSym) and the partial URL path indicated by inPathStr. (More about the callback specification frame in a moment.) It's strongly urged that you use a path beginning with your application symbol to prevent collisions in the path name space.

- nHTTPd:UnRegTranslator(inAppSym, inPathStr) removes the callback specification frame for the partial URL path indicated by inPathStr.

- nHTTPd:UnRegTranslators(inAppSym) removes all registered translators for the application whose symbol is inAppSym.

The callback specification frame provides a mechanism for nHTTPd to invoke a translator and has the same format as a regular communications callback specification. The only required slot is a CompletionScript slot (since all calls are asynchronous), containing a function that is passed, in order, these three arguments:

- inEp, the endpoint associated with the request

- inData, a frame containing the data received from the client

- inErr, an error code, or nil if the request began with no errors

The inData frame has at least four slots:

- **raw** is the original partial URL.

- **data** is the partial URL without the path.

- **path** is the path of the partial URL without the filename.

- **tag** is the partial URL with neither path nor any form data.

A **form** slot, if present, contains slots named for fields in the request; each slot contains a string bearing the value of the named field. Listing 1 shows what our second GET example above would expand to.

Listing 1. A processed URL frame

```
{
    raw: "/myApp/search.html?field1=hello&field2=world",
    data: "search.html?field1=hello&field2=world",
    path: "myApp",
    tag: "search.html",
    form: {
        field1: "hello",
        field2: "world"
    }
}
```

We'll probably also want to include a **_parent** slot in the callback specification, so that it can inherit from a useful context within the translator (see Listing 2).

OUR STATE MACHINE

Like most communications programs, our application can be represented as a state machine. The application can be in one of a finite number of states: while in one state, the application waits for an event, and the functionality of the program is encapsulated in the actions performed during a state change.

The nHTTPd state machine is shown in Figure 1. The arrows between states indicate transitions — where the program actually does something. For clarity, my implementation of this state machine uses symbols to denote each state, but using integer constants would save memory.

Listing 2. The translator callback specification frame

```
{
    _parent: self,
    CompletionScript := func(inEp, inData, inErr)
    begin
        inEp:Output("Why did you want " & inData.tag & "?", nil, {
            async: true,
            completionScript: func(inEp, inOpt, inErr)
            begin
                // Do something useful!
                ...
            end,
        }
    );
    inEp:Close();
    end,
}
```
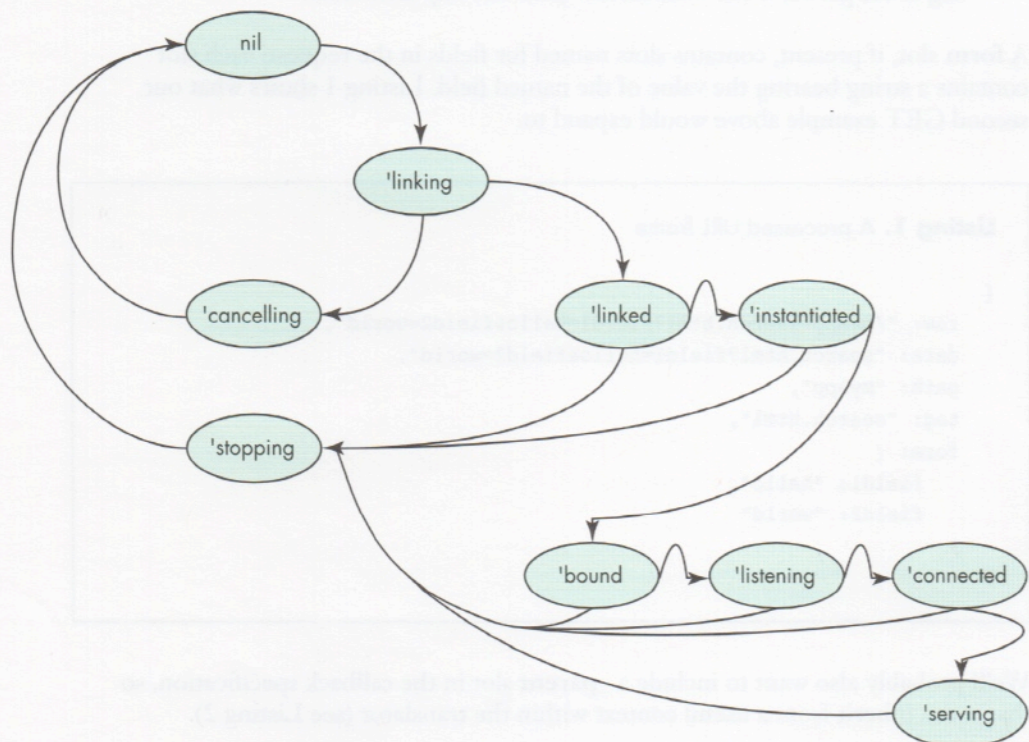

Figure 1. The nHTTPd server's state machine

Using a state machine provides many advantages during application development. Primarily, it's an organizational tool, allowing us to accurately plan and visualize what's going on inside our application. There's something very soothing (even if you're as bad a mechanic as I am) in imagining an application running as a set of gears clanking around, and it's a lot easier to follow than a spaghetti of method calls. There's also the benefit that most existing protocols are specified and implemented as

state machines, which makes porting somebody else's code a little easier. While the Newton is generally a difficult target for porting, porting from a state machine is significantly easier than porting from most other kinds of implementation.

Our state machine is a little odd, because there are really two separate machines running at once. We use a protoTCPServer (discussed in detail later) to manage the Internet link and provide endpoints listening on port 80 (the standard HTTP server port number). This state machine invokes an instance of protoHTTPServer, which itself is a smaller state machine that invokes your application.

GETTING A REQUEST

nHTTPd's protoTCPServer deals with all the details of establishing a link, as well as instantiating and listening on a TCP socket. The protoTCPServer has two methods:

- Start requests the link, instantiates an endpoint, and awaits a connection. When a connection is made, it's accepted, and the callback specification's CompletionScript is called with the resulting endpoint. The callback passed to Start is invoked whenever nHTTPd receives a connection on the monitored port.

- Stop cancels any open endpoints, disconnects and destroys them, and releases the open link. When we're done with the protoTCPServer, we use the Stop method to shut down all pending connections and close down the link. The callback for Stop is called when the teardown is complete and the link is released.

USING THE LINK CONTROLLER

Looking at the Start method gives a good idea of what's involved in using the Link Controller for the average application. The Link Controller APIs are all asynchronous, so as part of the arguments you must provide a context frame and the symbol of the completion script to be invoked (we'll see the details in a bit). Generally, you'll perform the following steps:

1. Your application will give the user a chance to configure the link by calling InetOpenConnectionSlip.

2. Your application will call InetGrabLink to request a link from the system.

3. Periodically as the link is established, the callback you provided to InetGrabLink will be invoked with status information. In your callback, you'll call InetDisplayStatus to notify the user of the progress of the link establishment.

4. Once the link is established, you'll use NIE endpoints as you would any other endpoint. You can also call InetGetLinkStatus to determine the current status of the link, to help determine when you're ready to begin using an NIE endpoint.

5. When you're through with the link, you'll call InetReleaseLink to end the connection.

The InetOpenConnectionSlip method presents the user with a slip for selecting an Internet service provider if no link is currently active. You pass three arguments to InetOpenConnectionSlip: the initial ID of the link to be offered (or nil to have the system select one for you), the frame to receive the callback message, and the name of the slot in the frame that contains the callback method to be invoked.

Once a selection is made, the callback is invoked with a single argument, either nil (indicating that no connection is to be made) or **'connect**. If a link is active, the slip

is not displayed, the existing link is used, and the callback is immediately invoked with the **'connect** symbol. Listing 3 shows how we present the slip along with a callback method to handle the user's choice (there are several callbacks — one in the listing for the user response, one passed to Start to be called in response to an incoming connection, and one to be called during link acquisition, as shown later in Listing 4).

Listing 3. Choosing a connection and handling the user's choice

```
Start := func(inCallbackSpec)
begin
   if fState then return;
   fState := 'choosing;
   InetOpenConnectionSlip(nil, self, 'mConnectSlipCb);
   fCallbackSpecs := Clone(kProtoCallbacks);
   fCallbackSpecs.fStart := inCallbackSpec;
end;

mConnectSlipCb := func(inAction)
begin
   if inAction = 'connect then
   begin
      fState := 'linking;
      fStatusView := BuildContext({_proto:
         GetLayout("protoInetStatusTemplate")});
      fStatusView:Open();
      InetGrabLink(nil, self, 'mGrabCb);
   end
   else fState := nil;
end;
```

Once the user has selected the preferred link mechanism, a status slip is created and presented to the user. The nHTTPd status slip is based on the Newton DTS sample Internet status slip, showing only a text view indicating the current status, along with a Stop button and a close box. Your status slip can contain anything you find appropriate, as long as it inherits from the protoInetStatusTemplate. Optionally, NIE can create a default status view for your application.

With the status view built and displayed, call InetGrabLink to request the link that was indicated by the user. InetGrabLink uses the same arguments as InetOpenConnectionSlip, although the callback itself takes three arguments: the link ID being used, the current status of the link as an NIE status frame (the only slot you need to worry about is the linkStatus slot), and a result code (which is nil for a successfully established link).

The process of acquiring a link can take a relatively long period of time, since the device may have to dial a modem and then execute a chat script to establish a PPP or SLIP stream before being able to conclude successfully. To keep your application informed, it invokes a progress callback periodically. In your callback, you can use the function InetDisplayStatus to keep the user apprised of status (see Listing 4).

InetDisplayStatus is a versatile function; depending on its arguments, it can do many things. You pass the current link ID, a reference to your status view, and a reference to the status frame from your callback. In turn, the function will show or hide the

Listing 4. The InetGrabLink status callback

```
mGrabCb := func(inLinkID, inStat, inErr)
begin
    fLinkID := inLinkID;
    if inErr then
    begin
        :mNotifyError("InetGrabLink", inErr);
        fState := nil;
    end
    else
    begin
        InetDisplayStatus(inLinkID, fStatusView, inStat);
        if inStat.linkStatus = 'connected then
        begin
            fState := 'linked;
            InetDisplayStatus(inLinkID, fStatusView, nil);
            fStatusView := nil;
            :mInstantiateAndBind();
        end;
    end;
end
```

status dialog or update the status indicator with the text of the current status as necessary. Table 1 shows the relationship between the arguments to InetDisplayStatus and the resulting behavior.

Table 1. InetDisplayStatus arguments and results

Second argument (view template)	Third argument (status)	Result
nil	nil	Returns a reference to a default status view, and opens the view for you
Template of a status view	Status frame (non-nil)	Uses the template as the status view and displays the status
A currently shown status template	nil	Closes and destroys the status view

Note that InetDisplayStatus can even create a status view for you, if you're so inclined. This status view is built on the protoInetStatusView and provides text indications of status to the user.

USING ENDPOINTS

Once the link is established, you're free to use endpoints to initiate TCP/IP connections. Our application uses an array of endpoints to service incoming requests; one endpoint is always listening for new connections, while other endpoints may be open responding to existing requests.

Our endpoints begin their life in the method mInstantiateAndBind, shown in Listing 5.

Note that setting up your endpoint works just as you'd expect. In addition, there's a host of compile-time utility functions that aren't officially supported but they're useful enough that I've included them with the program and summarized some of them in Table 2.

Listing 5. Creating and binding an endpoint

```
mInstantiateAndBind := func()
begin
   local myEp;
   if NOT fEndpoints then
      fEndpoints := [];
   myEp := {
      _proto: protoBasicEndpoint,
      _parent: self,
      fState: nil,
      ExceptionHandler: func(inExp)
         begin
            local myErr;
            if HasSlot(inExp, 'data) then
               myErr := inExp.data;
            if myErr <> kCancellationException then
               :Notify(kNotifyAlert, kAppName,
                  "Something bad happened - " & myErr);
            :mTearDown(self);
         end,
      EventHandler: func(inEvent)
         begin
            if inEvent.eventCode = kCommToolEventDisconnected then
            begin
               :Notify(kNotifyAlert, kAppName,
                  "The other side has disconnected." & myErr);
               :mTearDown(self);
            end
            else print("Unexpected event (" &  inEvent.eventCode & ")");
         end,
   };
   AddArraySlot(fEndpoints, myEp);
   try
      myErr := myEp:Instantiate(myEp, call kGetEndpointConfigOptions
                  with (fLinkID, kTCP));
   onexception |evt| do
   begin
      :mNotifyError("Instantiate", 0);
      return;
   end;
   if myErr then
   begin
      :mNotifyError("Instantiate", myErr);
      return;
   end;
   myEp.fState := 'instantiated;
```

(continued on next page)

Listing 5. Creating and binding an endpoint *(continued)*

```
    try
        myErr := myEp:Bind(
            call kINetBindOptions with (0, fPort), {
                _parent: self,
                async: true,
                CompletionScript: mBindCb,
            }
        );
    onexception |evt| do
    begin
        :mNotifyError("Bind", 0);
        :mTearDown(myEp);
        return;
    end;
    if myErr then
    begin
        :mNotifyError("Bind", myErr);
        :mTearDown(nyEp);
        return;
    end;
end
```

Table 2. NIE constant functions defined by include files

Function	Arguments	Purpose
kNumToHostAddr	inAddrArr	Turns an IP address into a string in the form "a.b.c.d"
kHostAddrToNum	inAddrStr	Turns a string address in the form "a.b.c.d" into the 4-byte IP address
kGetEndpointConfigOptions	inLinkID, inProtocol	Creates an options array for use when an NIE endpoint is instantiated
kINetBindOptions	inLocalPort, inUseDefaultPort	Creates an options array for use when an NIE endpoint is bound
kTCPConnectOptions	inRemoteAddrArr, inRemotePort	Creates an options array for use when an NIE TCP endpoint is connected
kUDPPutBytesOptions	inAddrArr, inPort	Creates an options array for use with a UDP endpoint when data is to be sent

The function kGetEndpointConfigOptions generates the options array necessary to use the NIE endpoint. There are options to request NIE (the **inet** option), specify the link kind (the **ilid** option), and indicate the desired transport (a flag indicating TCP or UDP is passed with the **itsv** option).

When binding your endpoint, you'll have a little more to consider. At the time your endpoint is bound, you'll pass different options depending on whether your endpoint is going to be listening (inbound) or connecting (outbound), and whether the endpoint is UDP or TCP. If you're using a UDP endpoint, or if your TCP endpoint is inbound, you'll need to specify a local port number. If your endpoint is an outbound TCP connection, NIE provides a randomly assigned port number for your endpoint. In the event that you need to set an option, you can use the function kINetBindOptions.

When would you want to specify a port? Most standard Internet applications have the notion of a reserved, or well-known, port. The server listens on the well-known port and clients specify that port as the destination port. Clients are free to use a random port for the source.

The function in Listing 6 creates an **ilpt** option frame containing the desired local port. We're invoking Bind asynchronously, and when it's complete, notification is achieved through the invocation of mBindCb.

Listing 6. Completing a Bind call

```
mBindCb := func(inEp, inOpt, inRes)
begin
   if inRes then
   begin
      :mTearDown(inEp);
      if inRes <> kCancellationException then
         :mNotifyError(inRes);
   end
   else
   begin
      inEp.fState := 'bound;
      try
         inEp:Listen(nil, {
               _parent: self,
               async: true,
               CompletionScript: mListenCb,
            }
         );
      onexception |evt| do
      begin
         :mNotifyError("Listen", 0);
         :mTearDown(inEp);
         return;
      end;
   end;
end;
```

From this point, what you do depends on whether you're expecting an incoming connection or initiating an outgoing request. Since I'm doing the former, I call my endpoint's Listen method, which instructs NIE to wait until an incoming request occurs. If you were calling Connect on a TCP socket to initiate an outgoing connection, this would be where you specify the destination address and port using the **irts** option — you can programmatically generate the appropriate arguments with a call to kTCPConnectOptions. In our case, the Newton waits until an incoming request is detected, and then notifies the application by calling the completion script originally passed to the Listen function (see Listing 7).

Listing 7. Processing an incoming request

```
mListenCb := func(inEp, inOpt, inRes)
begin
    if inRes then
    begin
        :mTearDown(inEp);
        if inRes <> kCancellationException then
            :mNotifyError(inRes);
    end
    else
    begin
        inEp.fState := 'listening;
        inEp:Accept(nil, {
            _parent: self,
            async: true,
            CompletionScript: mAcceptCb,
        });
    end;
end
```

The way we've written it, nHTTPd is an indiscriminate server — we accept connections from anywhere. If you wanted to do access control, or log where incoming connections were from, you could do it before we return control to the protoTCPServer's callback in mAcceptCb, shown in Listing 8.

At this point, control is passed to the client of the protoTCPServer, and the request phase of the connection has begun.

Listing 8. Accepting an incoming request

```
mAcceptCb := func(inEp, inOpt, inRes)
begin
    if kDebugOn then print("mAcceptCb");

    if inRes then
    begin
        :mTearDown(inEp);
        if inRes <> kCancellationException then
            :mNotifyError(inRes);
    end
    else
    begin
        inEp.fState := 'connected;
        inEp.Close := func()
        begin
            if kDebugOn then print("Close");
            :mTearDown(inEp);
        end;
        fCallbackSpecs.fStart:?CompletionScript(inEp, nil, nil);
    end;
end
```

That's the basics of the application — creating endpoints, accepting configuration from the user, waiting for requests, and accepting remote connections. We still need to read serial data to process URLs, do something useful with them, and write out response data. But first, let's do some more work with the DNS.

USING THE DOMAIN NAME SERVICE

For nHTTPd to be truly useful, it needs to track and log the host names generating incoming requests. We could simply log the IP addresses of incoming hosts, but these would mean little to the user. Instead, nHTTPd uses the DNS of NIE to ascertain the host name of incoming queries.

The DNS, like the Link Controller, uses asynchronous calls to do its work. You'll pass information to one of the DNS functions and receive a response through a completion script. The DNS provides several global functions for your use:

- DNSGetAddressFromName, which converts a host name string to its IP equivalent
- DNSGetMailAddressFromName, which discovers the IP address for a mail server given a host name in a particular domain
- DNSGetMailServerFromName, which returns the name of the mail server given a host name
- DNSGetNameFromAddress, which returns a host name string given the IP address
- DNSCancelRequests, which cancels all requests associated with a specific context

The first four functions take three arguments: the key for the data to be retrieved, the client context, and a symbol denoting the callback. The cancellation function takes only a client context and a callback symbol. For all but the last function, the callback receives a results array and result code. The results array contains result frames, each with one or more of the following slots:

- type, the result type (kDNSAddressType or kDNSDomainType)
- resultDomainName, a string containing the resulting domain name
- resultIPAddress, an array containing the 4-byte IP address

Listing 9 shows a code fragment that determines the host name for a particular IP address and logs it to a soup.

RECEIVING INPUT

Now that we have a connection and have logged the domain it came from, we need to do the work of reading the URL request. Getting data into an application is done as with any other endpoint: set up an input specification and wait for your data to come to you. The input specification used by nHTTPd is simple, as shown in Listing 10.

If you've done Newton communications work before, this is nothing new. If you haven't, it's worth taking a moment to consider the notion of an input specification.

Rather than providing a traditional stream for input, the Newton OS provides the notion of an input specification, which describes the format of the information your application expects. You build an input specification (or "input spec" for short) using

Listing 9. Translating a name to an address and logging it

```
DNSGetAddressFromName(fDestAddr, self, 'mDNSResultCb);

mDNSResultCb := func(inResultArr, inResultCode)
begin
   if inResultCode then
   begin
      // Rats. It failed.
      print("DNS lookup failed because of " && inResultCode);
   end
   else
   begin
      local myAddrFrame;
      local myHandyAddresses := foreach myAddrFrame in inResultArr
                                  collect myAddrFrame.resultDomainName;
      fLogSoup:AddToDefaultStoreXmit( {
            hostnames: myHandyAddresses,
            request: fRequestString
         },
         kAppSymbol);
   end;
end;
```

Listing 10. The input specification

```
local myInputSpec := {
   form: 'string,
   termination: { endSequence: unicodeCR },
   filter: {
      byteproxy: [
         { byte: kUnicodeBS, proxy: nil },
         { byte: kUnicodeLF, proxy: nil },
      ]
   },
   inputScript: func(inEp, inData, inTerm, inOpt)
   begin
      local myResult, myClient;

      // Optimization - why bother if nobody's registered?
      if Length(fRegistry) = 0 then
         :mReportBogusAndClose(myEp);

      if StrLen(inData) = 0 then return;
      // Change stuff like %20 to ' '.
      :mFilterPercentEscapes(inData);
      // Break out the data frame for the application callback.
      myResult := :mCreateArguments(inData);
      foreach myClient in fRegistry do
         if StrEqual(myClient.path, myResult.path) then
            break;
```

(continued on next page)

things like the class of the incoming data, its length or an array of possible termination characters, how long to wait for input, or communications flags that denote the end of input. The Newton uses the input spec to watch for input in the background while your application is running. Once the conditions of the input spec are met, its inputScript is invoked; if the input spec is never satisfied, the CompletionScript may be invoked to notify you that the input spec was never filled. This will happen if you timeout on a receive, or if the endpoint is closed while you're expecting input. As your application runs, it can change which input spec is active with the endpoint method SetInputSpec. This aspect of the Newton communications model makes writing applications based on state machines incredibly easy, and porting most code from stream-based environments painfully difficult. For more on communications state machines, see "Use a State Machine."

There's an exception to input handling that you should know about, even if you never use it. When you use a TCP endpoint, expedited data isn't passed to your application through the normal input spec means. Instead, any expedited data is sent as an event to your endpoint's event handler. Listing 11 shows such an event handler (the method mTearDown, called in this event handler and many other places, is shown later).

USE A STATE MACHINE

When planning an application, take pains to be sure you've developed a good state machine to represent it. Effort during the design phase will be repaid a thousandfold when your application's communications code is almost entirely an array of input specifications and their scripts. If you're new to working with state machines and communications, check out the Newton DTS sample CommsFSM, which has enough to get you started.

If you're porting code, take heart, and do the same. Virtually all protocols can be well represented by a state machine, and if you do so you'll find it's easier to implement. At all costs avoid simulating a UNIX®-style stream with an endpoint — it's expensive, and you'll almost never need the functionality of a byte stream.

Listing 11. Receiving expedited data in an event handler

```
EventHandler:func(inEvent)
    begin
        if inEvent.eventCode = kCommToolEventDisconnected then
        begin
            GetRoot():Notify(kNotifyAlert, kAppName,
                "The other side has disconnected.");
            :mTearDown(self);
        end
        else if inEvent.eventCode=kEventToolSpecific then
            if inEvent.data < 0 then
                print("Got an error (" & inEvent.data & ")");
            else
                print("Expedited data! The byte was" & inEvent.data);
        else
            print("Unexpected event (" &  inEvent.eventCode & ")");
    end,
```

WRITING A RESPONSE

By now, I hope that you've guessed how to send data: you use the endpoint's Output method, shown in Listing 12.

Listing 12. Sending data

```
inEp:Output("Why did you want " & inData.tag & "?", nil, {
    async: true,
    completionScript: func(inEp, inOpt, inErr)
    begin
        // Do something useful!
        ...
    end,
});
```

NIE provides two options of interest for output. The first is the expedited data option, which can be used to set the expedited flag on a TCP packet. Expedited data is often used, for example, to signal the other end of the connection that it should cease transmitting data immediately. An expedited data byte is denoted with the **iexp** option, with a frame like the one shown in Listing 13 (where inByte is the byte to be expedited).

The other option is required for UDP communications — the address and port of the destination. This is required when output operations are performed, because UDP is connectionless (and you didn't specify the remote address during the call to your endpoint's Connect method). More on that in a bit.

DISCONNECTING WHEN DONE

Tearing down and cleaning up is exactly the same as with a standard endpoint: cancel any pending operations, disconnect, unbind, and then dispose of your endpoint. The protoTCPServer uses the method mTearDown, in conjunction with an endpoint's state variable fState, to determine the appropriate behavior (see Listing 14).

Listing 13. An expedited data frame

```
{
    label: "iexp",
    type: 'option,
    opCode: opSetRequired,
    result: nil,
    form: 'template,
    data: {
        argList: [ inByte ],
        typeList: ['struct, 'byte ],
    },
}
```

Listing 14. Disconnect and cleanup

```
mTearDown := func(inEp)
begin
    if inEp.fState = nil then
        print("We do nothing.");
    else if inEp.fState = 'instantiated then
        :mUnBindCb(inEp, inOpt, inErr);
    else if inEp.fState = 'bound then
        :mDisconnectCb(inEp, inOpt, inErr);
    else if inEp.fState = 'listening then
    begin
        inEp:Cancel({
            _parent: self,
            async: false,  // avoid async race condition
            completionScript: func(inEp, inOpt, inErr)
                :mDisconnectCb(inEp, inOpt, inErr)
        });
    end
    else if inEp.fState = 'connected then
        inEp:mDisconnect();
    else if kDebugOn then print("Endpoint is already closing!");
end


mDisconnectCb := func(inEp, inOpt, inRes)
begin
    try
        inEp:Unbind({
            _parent: self,
            async: true,
            completionScript: func(inEp, inOpt, inRes)
            begin
                :mUnbindCb(inEp, inOpt, inRes);
            end;
            }
        );
    onexception |evt| do nil;
end;
/* and the other callbacks look similar...*/
```

There's not much to discuss about the teardown procedure, except a couple of hard knocks you may get when actually using NIE endpoints (or any other kind):

- It's much better form to track your endpoint's state using a separate value than to retrieve it with the endpoint's State method, so that the application can use the state in other areas such as user interface or as part of the overall state machine.

- Be sure you wrap the endpoint's teardown methods in exception handlers. Although nothing's supposed to go wrong during an endpoint's teardown and cleanup, you never know, and you don't want an application throwing exceptions to your end users. This is especially true if your endpoint is being torn down after something bad has already happened to an open connection.

Unlike other endpoints, once you've disposed of an NIE endpoint, there's still a link hanging around that you'll need to get rid of. You do this with a call to the Link Controller's InetReleaseLink function, which will drop the link only if no other application is currently using it. Our protoTCPServer invokes this method after disposing of its last endpoint, as shown in Listing 15.

InetReleaseLink takes the same arguments as InetGrabLink — the link ID, a reference to a status view if appropriate, and what to put into the status view (or nil to conceal it entirely). Although it's obvious, be sure your application has the same number of InetGrabLink calls and InetReleaseLink calls. It's embarrassing to leave the link open or clobber it on another application.

Listing 15. Disposing of endpoints and releasing the link

```
mUnbindCb := func(inEp, inOpt, inErr)
begin
   try
      inEp:Dispose();
   onexception |evt| do nil;
   SetRemove(fEndpoints, inEp);
   if Length(fEndpoints) = 0 then :mReleaseLink();
end;


mReleaseLink := func()
begin
   fState := 'stopping;
   InetReleaseLink(fLinkID, self, 'mReleaseCb);
end


mReleaseCB := func(inLinkID, inStat, inErr)
begin
   if inErr then
   begin
      :mNotifyError("InetReleaseLink", inErr);
      fState := nil;
      fEndpoints := nil;
      fCallbackSpecs.fStop:?CompletionScript(self, nil, inErr);
      fCallbackSpecs := nil;
   end
```

```
    else
    begin
        if inStat.linkStatus = 'idle then
        begin
            InetDisplayStatus(inLinkID, fStatusView, nil);
            fState := nil;
            fEndpoints := nil;
            fCallbackSpecs.fStop:?CompletionScript(nil, nil, nil);
        end;
    end;
end
```

In addition to releasing the link when you're done with it, you'll want to release it if you won't be using it for a long period of time (over 15 minutes or so) to avoid battery drain from an internal modem or the like.

WHAT IF THIS WERE UDP?

The Newton endpoint model is connection-oriented; it doesn't directly work with a connectionless protocol such as UDP. Invocation of a UDP endpoint is essentially similar to a TCP endpoint, except that you never indicate the destination IP and port at the time you connect. Instead, you'll indicate them during your call to the endpoint's Output method, using the **iuds** option.

When performing input or output with UDP, be sure you always indicate packet boundaries. You do this by forcing a packet boundary on output, by including the kPacket and kEOP flags in your output and input specifications (see Listing 16).

Table 3 indicates which options you'll set at which times for which kinds of connections. You can use this as a summary to help you in creating your own applications.

Listing 16. Using packet boundaries for UDP

```
local myInputSpec := {
    form: 'string,
    termination: {useEOP: true},
    rcvFlags: kPacket,
    rcvOptions: {
        label: "iuss",
        type: 'option,
        opCode: opGetCurrent,
        result: nil,
        form: 'template,
        data: {
            arglist: [
                [ 0,0,0,0 ],    // Host address
                0,              // Host port
            ],
```

(continued on next page)

Listing 16. Using packet boundaries for UDP *(continued)*

```
        typelist: [
            ['array, 'byte, 4],
            'struct,
        ],
    },
    inputScript: func(inEp, inData, inTerm, inOpt)
    begin
        // Do something with the data.
        ...
    end,
    completionScript: func(inEp, inOpt, inRes)
    begin
        if inRes <> kCancellationException then
        begin
            print("Input spec saw error " & inRes);
            :mTearDown(inEp);
        end;
    end,
};
fEndpoint:SetInputSpec(myInputSpec);
fEndpoint:Output("Hello world!",
    call kUDPPutBytesOptions with (fAddr, fPort), {
    async: true,
    sendFlags: kPacket+kEOP,
    completionScript: func(inEp, inOpt, inErr)
    begin
        // Do something useful!
        ...
    end,
});
```

Table 3. NIE options and their uses

Option	Provided by	Used with	Inbound/ outbound	Purpose
inet	kGetEndpointConfigOptions	Both	Both	Specify NIE at instantiate
ilid	kGetEndpointConfigOptions	Both	Both	Specify link ID at instantiate
itsv	kGetEndpointConfigOptions	Both	Both	Specify either TCP or UDP at instantiate
ilpt	kINetBindOptions	TCP	Inbound	Specify local port for inbound TCP during bind
ilpt	kINetBindOptions	UDP	Both	Specify local port for UDP during bind
irts	kTCPConnectOptions	TCP	Outbound	Specify remote IP and port for TCP during connect
iexp	N/A	TCP	Both	Specify to expedite data during output
iuss	kUDPPutByteOptions	UDP	Both	Specify remote host and port during UDP output

GO OUT THERE AND NEWTONIZE THE INTERNET!

With the exception of link-level management, using NIE is the same as using any other kind of Newton endpoint. The NIE's Link Controller and DNS use new global functions to provide support for a unified link-level interface and domain name service. The design of NIE strongly encourages asynchronous programming techniques.

Using NIE to port existing applications is easy, as is writing new ones. With a little work, your application can soon be merging on the infobahn among the desktop machines already cruising the Net.

Newton Q & A: Ask the Llama

Q *I recently saw the announcement of two new Newton devices — the MessagePad 2000 and the eMate 300 — and I have a few questions. First, will my application run fine?*

A If your application was written correctly for Newton OS 2.0, for the most part it will work fine on Newton OS 2.1, which is what the new devices use. "Correctly" here means that you call only documented functions, including platform file functions where appropriate. It also means that your application works with different screen sizes by using GetAppParams in combination with minimum and maximum sizes.

Q *What are the most important things to check in my application to ensure that it's completely compatible with the new devices?*

A There are four broad areas to check: speed, screen size, views, and PC cards.

Speed. The new units are faster than the current ones. In the case of the MessagePad 2000, the difference is quite large. Unfortunately, it's possible for some things to be too fast. The new OS takes care of several speed issues for you — scrolling, for example — but there are still some areas you should check.

Check those places where you're doing repeated actions from a viewClickScript. A typical usage would be a button that will continually perform an action as long as the user presses it. If you use a loop in a viewClickScript to do the repetition, you may find that there are too many repetitions or that the repetitive action occurs too quickly. The same problem can occur if you don't use the scrolling API provided by the system, as scrolling is one area that has deliberately been slowed down on the MessagePad 2000.

Loops can also cause problems when they're used for timing. In general, you shouldn't use loops for this purpose; use Ticks instead. If you must use a loop, set the counter for that loop based on a known reference like Ticks or TimeInSeconds.

Operations that used to be long enough to require user feedback may now happen fast enough that no feedback is required. This can happen in two places:

- You may have been deliberately turning on the busy box. The result can be a busy box that flashes very briefly, which can be distracting.

- You may have implemented progress bars using DoProgress, protoStatus, or even a protoGauge. Try removing the progress indicator and checking whether the operation is now fast enough. Note that most of the progress indicators take time to draw and update — in some cases significantly longer than the time to do the operation for which the progress is being displayed.

Screen size. Be sure your application works properly in both portrait and landscape orientation, with the button bar on both the left and the right. In addition to the size of your overall application on the screen, check areas where you use complex justification or dynamic allocation of view children. Check that the children are correctly aligned — and that there are the correct number of children.

Also, be sure your borders don't go outside the application area. Note that borders are drawn outside the view. You can find your global bounds with the borders by calling GlobalOuterBox.

Views. The most likely problem related to views is assuming that the top left of the application area is always at (0, 0) in global coordinates. This is no longer the case since the button bar, which is 46 pixels wide, can be on the left side of the screen. A typical place for this problem to occur is in a viewClickScript where you do something with a point that's actually 46 pixels out from where you expect it to be. Rotate to landscape orientation and put the button bar on the left; then try tapping on the right side of your application. One sure way to cause problems is to forget to send SyncView to your base application view after a ReorientToScreen message has been sent.

Another possible view problem is that any view can be the keyView. Don't assume that the keyView can accept text input; in particular, don't use calls like SetValue to jam the text slot (which may not be there). You may want to check the class of the keyView.

PC cards. You should check that your application works when two PC card slots are being used for storage. Search for the following pieces of bad code:

```
GetStores()[1];   // BAD -- 0 is the only number documented to work

Length(GetStores());   // doesn't tell you the number of PC cards
```

If you have code like this, you'll have to change it. The first case — assuming there's a store at the second position in the array — is not a good idea. Even if there's a store at that position, it may not be the same store that was there the last time you checked. Also note that the positions in the array do not correspond to physical PC card slot positions. The second case can fail for similar reasons. That is, checking the length of the array returned by GetStores doesn't tell you how many PC cards are currently installed.

Along the same lines, if you're still using the action (routing) picker to move items to the card, you should change to using the filing interface. Also, make sure that you use the FileThis method to move items to different stores, and that you look at the arguments provided by FileThis; some application code seems to assume that there are only two stores. For adding soup entries, remember to call AddToDefaulStoreXmit instead of using the store directly.

You might also encounter a problem with endpoints that could use PC card modems. To set up your modem endpoint, call MakeModemOption, which will construct the correct options based on the user modem preferences and available PC card modems.

Q *Are there any features that are important to support in the new devices?*

A The most important thing is to make sure your application works. After that, there are some important updates you can make to support features in the new devices.

First, make sure your application can be dragged, assuming of course that it's not full-screen in all orientations. You can do this by changing your base view to a protoDragger with either some sort of status bar (preferably newtStatusBar) or

a protoLargeCloseBox. Remember to make sure the borders of the protoDragger will be contained in the application area; you can use GlobalOuterBox to check this. And while you're at it, you can check that your application will handle any screen size.

You may also want to see whether you can improve your use of screen space. Note that your major layouts (for example, detail and overview) don't have to be the same size. The Names application is a good example of this.

Another important feature to support is the use of the keyboard. Add the required keyboard commands to your application. As of this writing you can find this information in the "User Interface Guidelines for Newton OS 2.1 Keyboard Enhancements" document. If your application is based on NewtApp, most of this work is done for you; otherwise you'll have to add almost all the keyboard commands yourself. Once you've supported the required set, add other commands that make sense for your application. Don't forget keyboard navigation in your overview.

A related feature that's good to support on both Newton OS 2.0 and 2.1 devices is conditional display of embedded keyboards. You can use the KeyboardConnected global function to check whether a keyboard is connected; if it is, don't display embedded keyboards unless they're highly specialized.

If you're using the infrared (IR) communications tool, you should use IrDA if possible. This will give you a faster transfer rate and a much more robust protocol. If your application might be communicating with older units, be sure to give your users a choice of IR connection types, since older units can only use the ASK protocol. Newton OS 2.1 still supports all the IR options from 2.0. Note that using the action menu to beam information will do the right thing.

You should also take advantage of the grayscale feature, by using the new RGB-based gray shades (that is, kRGB_Gray1 through 15) instead of the dithered gray patterns. Dithered patterns are usually specified as vfGray, vfLightGray, and so on. You can also change your own patterns to use grayscale. Although the dithered patterns still work, the true gray RGB shades look a lot better. You'll want to wrap the specification in a check to make sure that grayscale is available. Naturally you'll want to update important parts of your application to use grayscale — for example, your splash screen and Extras icon.

Finally, if you're targeting the eMate and the education market, you should update your application for multiuser mode. This could be an extensive change, since you'll have to modify your interface and the names of all soups that you save.
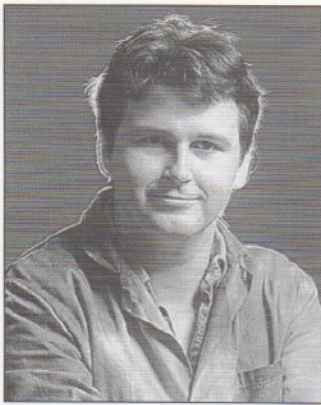
THE VETERAN NEOPHYTE

Digital Karma

JOE WILLIAMS

So shines a good deed in a weary world.
   — Willy Wonka

We all shine on.
   — John Lennon, *Instant Karma*

We at Delta Tao Software (creators of Spaceward Ho!, Strategic Conquest, and Eric's Ultimate Solitaire) have been working on Clan Lord, a network game that may someday have thousands of players exploring and colonizing an electronic landscape. We want a harmonious online world that's enjoyable for every player, and we've come up with a system that encourages this, without authoritarian overtones — or at least we hope it will. This column looks over some of our system's mechanics, repercussions, and possible applications beyond gaming, and gives some food for thought for your own projects.

HOW CLAN LORD USES KARMA

Clan Lord is a big Mac network game. (No, not a Big Mac network game; we'd hate getting sued by McDonald's!) Network games are nothing new to us, but now, with the massive proliferation of the Internet, we want to do a truly epic game.

Clan Lord doesn't fall under the traditional definition of a game: there's no end, and there are no winners and losers. It's more like a complete world, and each player is a member of an online society. Online societies naturally develop their own customs, ethics, and morals, just as other groups do. Our goal is to have the world be enjoyable — a pleasant place to meet and interact.

The key to a good party is inviting the right people. Some people enhance their environment. In an online world, they answer questions, help people, and encourage others to exhibit proper behavior. We'll call these people "good."

Some people do their best to ruin the party for everyone. In an online world, they ridicule, shout, and abuse. They're often found saying things like "Bob Dole is a lemonhead!" and "HOWARD STURN RULEZ!" We'll call these people "bad."

On a service like America Online, the worst people are eventually ejected. But good people generally receive only personal satisfaction from their sometimes considerable efforts. That the Internet information-sharing exchange works so well is a marvel that speaks well of (oft-ridiculed) human nature, but it could work even better. Our goal is to increase the ratio of good people to bad and the likelihood of good behavior.

Without external controls, games like this (usually called MUDs, for Multi-User Dungeons) tend to devolve into hack-and-slash slugfests. New players join in, only to find themselves repeatedly killed by more experienced players. Discouraged and humiliated, they abandon the game. This problem is usually "fixed" by rule changes that make it impossible to attack newer players, or by threats from the game developers to eject (bad) people who hunt other players. These solutions often cause as many problems as they solve. Established games are most successful when there's a core of (good) veterans who encourage and protect the "newbies."

So we want lots of good people, and not too many bad ones — but how do you tell one from the other? A host could moderate, flagging people one way or the other. But this is subjective, and it reeks of authoritarianism. It's also too much like work. We came up with a painless, automatic solution for our game: digital karma.

For every day you spend in Clan Lord, you can give 100 karma points to other players, as either good karma or bad karma. If someone solves a problem for you, or says something you agree with, or even gives a friendly word to a stranger, you can send them some good karma. If someone insults, curses, lies, or bugs you in any way, you can send them some bad karma. You can't give karma to yourself, and you can't change (or respend) the karma other people give you.

Over time, this karma adds up to a number telling how "good" a person is. People with good karma earned it with good words and deeds, and people with bad karma

JOE WILLIAMS is founder and president of Delta Tao Software, the self-proclaimed "coolest company in the world." He writes a rambling daily e-mail column to which you can subscribe by sending "subscribe joedeltatist" in the body of your message to majordomo@outland.com. Joe has +47 karma. •

earned it by being annoying and antisocial. In Clan Lord, some places are accessible only to people with good karma, while people with bad karma may have to fight their way out of "Hell" when they die. Organizations of players (called *clans*, of course) also have karma ratings, just by summing the karma of their members.

BUT DOES IT WORK?

There's never been a digital karma system, so we spent a lot of time worrying that it might cause as many or more problems than it solves. What if bad people abuse the system to make themselves appear good? What if good people go unrewarded, become disillusioned, and go bad or — worse — quit altogether?

It might be that, since the total sum of karma is large, individual abuses would get averaged out. Since each person gets 100 karma points to distribute per day, thousands of people means hundreds of thousands of karma points moving around. However, since the average karma received by each person is 100 a day, an individual can concentrate his or her karma to have a substantial influence on specific other individuals.

For example, Ma and Pa Barker spend a day tormenting Elvis. (See Figure 1.) Elvis gives each of them 50 bad karma. But Ma gives Pa 80 good karma for holding Elvis down while she kicks him. And Pa gives Ma 80 good karma for doing such a good job tying Elvis's hands to his ankles. And they each give Elvis 20 bad karma for whining so much. When lovely Rita (Meter Maid) comes by, she might cart Elvis (with 40 bad karma) off to jail for tormenting the obviously virtuous Barkers (with 30 good karma each). "You must have really made them angry," she murmurs as she slips on the handcuffs.

+80 from Pa
−50 from Elvis
+30 total

+80 from Ma
−50 from Elvis
+30 total

−20 from Pa
−20 from Ma
−40 total
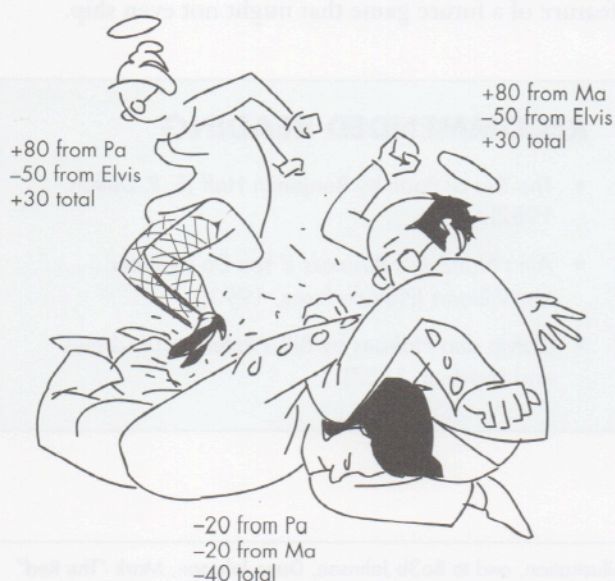
Figure 1. Karma distribution

This sort of thing is certain to happen, but it's likely that in the long run the Barkers' evil ways will catch up with them. Elvis can relate the tale to his friends George, John, Paul, and Ringo, who together can inflict more bad karma on those nasty Barkers than they know what to do with. What's more, the Barkers have gone on to annoy yet another innocent bandsman who can inflict some bad karma of his own. In the long run, the Barkers have to spend as much time pleasing people (even each other) as hurting them to keep their karma in the black.

What about the problem of do-gooders not getting rewarded? Johnny B. wanders in to find helpless Elvis, tied up and bruised from his run-in with the Barkers. He cuts away the bonds, applies first aid, and gives Elvis some spending money. Elvis, instead of bestowing good karma on Johnny B., inflicts bad karma on the Barkers. Johnny B. finds himself with no better karma, despite his afternoon of good deeds. Maybe next time he'll be less inclined to help the helpless.

But probably not. He still gets all the intangible karma he would have gotten before our system was in place. Elvis is still grateful. Johnny B. will, over time, get plenty of good karma for his benevolent activities. In addition to getting the unquantifiable benefits that come with doing a good turn daily, he'll get an occasional reward of good karma. He'll probably be even more likely to do good deeds than he was before.

Now, the worst case. Ozzy, after watching *Natural Born Killers*, decides it would be fun to see just how much bad karma he can rack up. He traipses through the countryside setting fire to outhouses, pushing grandmothers down stairs, and biting the heads off of innocent rodents. He's bad to the bone. He racks so much bad karma that he grows horns and hooves. But he doesn't care — he's going for the record. The baddest cat ever, and our little system gives him the numbers to prove it. Aren't we just egging him on?

There are always going to be a select few who delight in infamy. Perhaps a few of those could be persuaded to form vigilante groups, hunting the Most Wanted of the Bad Karma Boys. Just as many folks are going to go for the evil-punishing record as for the bad karma record. We'd turn those of Ozzy's frame of mind against each other. His worst enemy is his own kind.

WHERE DOES IT GO FROM THERE?

Besides modifying behavior, karma studies can identify trends, and possibly give warnings of societal problems. The ratio of good karma to bad karma is an indicator of how happy the society is as a whole. A sudden drop in that ratio could be an early symptom that something is wrong, giving us a chance to nip the problem in the

bud. If we do our job, that ratio should see a gradual increase over time, as we weed out things that make people unhappy.

We'll no doubt also see some interesting statistics. I'm curious to see whether people who send lots of bad karma are the same people who receive it, and vice versa. I've certainly always suspected that that's how it works in life. Any complex economic system, like our described karma system, is largely unpredictable. Who's to say what kind of karma wars might erupt? A system like this will have consequences nobody anticipates. It's fun to do thought experiments, as we've done here, but the fact is we won't really know until afterward.

And there are lots of questions for which we hesitate to guess the answers: Is it better to give karma feedback to recipients immediately, so as to help them modify their actions appropriately? Or is it smarter to delay this information and make it anonymous, so that repercussions and threats won't influence its delivery? Who knows? These questions aren't likely to be answered without lots of testing and experimentation, but once that's done, we're likely to have a more pleasant and predictable online society.

We've been talking about digital karma in terms of our game, but that doesn't need to be where it ends. Newsgroups and bulletin boards would certainly benefit by having more good people and fewer bad. You could screen out bad-karma messages or the people that post them. And people who post would get measurable feedback on how helpful their posts are, without having to sift through a lot of noise.

INSTANT KARMA'S GONNA GET YOU

Predicting how computers will be used has never been easy. In the fifties, people thought computers would eventually calculate missile trajectories and save the world from the evil communists. In the seventies, the advent of the personal computer led Marketing people to reveal the true possibilities: we could balance our checkbooks and organize our recipes.

In fact, technology has affected nearly every aspect of our lives — or my life, anyway. But technology hasn't had much effect on social behavior, which is probably the one area where we need it most. With the possible exceptions of soothing fish tank screen-savers and "simulations" that show the perils of evildoing by

exploding the spleens of undead Nazis, computers have made even less impact on social mores than Dungeons and Dragons.

Not every piece of software, and certainly not every game, can (or should) try to have social relevance. But it's sure fun to keep one's eyes open for the possibility. That's how the Mac was created in the first place: computer geeks trying to change the world.

Naturally, karma already exists outside the online world — it's just intangible and unquantifiable. Do-gooders (darn them!) tend to get what's coming to them; I'm more likely to go out of my way to help someone who has been helpful to others in the past. But it would sure be nice to be able to quantify this karma, so that we could get an idea of someone's "goodness" upon first meeting. Of course, it's possible that trying to quantify karma like this, putting numbers to it, will just create a sort of "behavioral economy," whereas the "real" karma, the intangible kind, will continue to exist independently, however we keep score.

Wouldn't it be nice to be able to distribute karma in the real world? If we tracked it properly, karma would be more important than credit — as it should be. Someday, with technology's help, we might be able to point a remote control and push a couple of buttons to reward that nice librarian or those kids that wrote that great game. Or finally do something about that granny in her fume-spewing Pinto, that dog-kicking punk, or the neighbor who mows his lawn at five-thirty in the morning.

Indeed, digital karma could revolutionize society — or maybe it's just a thought experiment about a potential feature of a future game that might not even ship.

RECOMMENDED READING

- *The Tao of Pooh* by Benjamin Hoff (E. P. Dutton, 1982).

- *Ain't Nobody's Business If You Do* by Peter McWilliams (Prelude Press, 1993).

- *Calvin and Hobbes* by Bill Watterson (Andrews and McMeel, 1987).

Macintosh Q & A

Q *I noticed in QuickTime 2.5 that there are new selectors in the Movie Import API that aren't described in Inside Macintosh: QuickTime Components. Although I found some information in Technote QT 04, "QuickTime 1.6.1 Features," I haven't been able to locate any information about the kMovieImportGetFileTypeSelect and kMovieImportDataRefSelect selectors. Can you tell me something about these?*

A The kMovieImportGetFileTypeSelect and kMovieImportDataRefSelect selectors were added to support some features that were under investigation with the QuickTime for Netscape plug-in. While these selectors are supported by some of the Apple movie data import components, they don't provide any new functionality and there's no reason to consider implementing them in your movie data import component at this time.

Q *My application plays four QuickTime movies simultaneously from a Director project. Each of the movies has a single music track with no other video or sound tracks, and two of the movies use more than one instrument. The Director project allows the user to control the volume level of each movie independently. The application works great on the Macintosh with QuickTime 2.1, but under Windows with QuickTime 2.1.2 only one music track plays at a time. Is it possible to hear all four music tracks at once under QuickTime for Windows 2.1.2?*

A You can do live mixing of your four QuickTime movies only if your Windows system has four MIDI output devices. Most systems have only one. All Windows applications suffer from this limitation unless they're clever enough to mix the tracks on the fly, but none seem to do this.

For now, you must pre-mix the four music tracks from the four movies into one music track in one movie. You won't be able to do live mixing unless you write your own MIDI sequencer.

Q *Can I play a compressed WAVE file on the Macintosh?*

A Yes. You can use the Sound Manager to play a compressed WAVE file on the Macintosh, but how easy it will be can vary greatly, depending on the type of compression.

If the WAVE file is compressed using μlaw, your program will have to use either SndPlayDoubleBuffer or SndDoCommand with bufferCmds to play the sound (QuickTime cannot currently be used). This means that you'll have to parse the WAVE header yourself and interact with the Sound Manager at a lower level than is possible by simply calling SndStartFilePlay.

Currently, you can't play an IMA-ADPCM compressed WAVE file as easily as a 'ulaw' compressed WAVE file, because the data stream of a sound compressed with Windows' IMA-ADPCM compressor differs from the data stream of the same sound compressed with the Macintosh's IMA compressor. To play an IMA-ADPCM compressed WAVE file on the Macintosh, your application will first have to decompress chunks of the sound into memory, then have the Sound Manager play those chunks. You can do this either by calling SndPlayDoubleBuffer or by using bufferCmds. Of course, if the uncompressed sound fits completely into memory, you can simply use SndPlay on that one uncompressed buffer.

To play a WAVE file that uses a custom compression algorithm, you can either write your own 'sdec' (sound decompressor) component or simply have your

program decompress the sound itself. As before, if you can fit the decompressed sound in memory, you can use any Sound Manager routine to play it. If you can't, you'll have to decompress it in chunks and use SndPlayDoubleBuffer or a bufferCmd to play each chunk. If you write your own 'sdec', of course, you can use any Sound Manager routine that will play an arbitrarily compressed sound, but be sure to indicate that the sound is compressed with your compressor so that the Sound Manager will know to call your 'sdec'.

Q *How do I make a sound that will play on both Macintosh and PC computers?*

A This is actually easy, as long as you don't want to play compressed sounds. We recommend that you use WAVE files for both systems, since they'll play easily on the Macintosh. (See the previous answer for some tips to help you play WAVE files on the Macintosh.) Note, however, that there are many other formats that will work, including AIFF and QuickTime.

Q *How can I access the "Set Utilities Pattern" pattern? (This pattern, used as a background by certain system utilities like Find File, is normally set by holding down the Option key in the Desktop Patterns control panel.)*

A The Desktop Patterns control panel uses resources of type 'ppat' to store both desktop and utilities patterns. The 'ppat' resource is stored in the System file in your System Folder; the desktop pattern has an ID of 16 and the utilities pattern has an ID of 42. Since this isn't documented, it could be subject to change, so you should be careful when using it.

Here's a snippet of code that shows how you can get the utilities pattern and then draw with it:

```
PixPatHandle    ppatHandle;
Rect            destRect;

ppatHandle = (PixPatHandle) GetPixPat(42);
if (ppatHandle != NULL) {
    SetRect(&destRect, 15, 125, 197, 164);
    FrameRect(&destRect);
    FillCRect(&destRect, ppatHandle);
    DisposePixPat(ppatHandle);
}
```

Q *I discovered that if I hold down the Command key and click in the size box of a window, I can make the window bigger than the width I pass into GrowWindow. The size box works as expected when the Command key isn't used. This seems to happen in every application I've tried. I haven't been able to find any documentation discussing the relationship between the Command key and the size box. I really need to limit the width of my windows. Why is this happening and how can I work around it?*

A Back in the old days when the Macintosh had a 9-inch screen (384 x 542 pixels), a lot of developers didn't follow Apple's guidelines for window sizes and hard-coded the sizeRect given to GrowWindow based on this small size.

When the Macintosh II was introduced in 1987, Apple engineers felt it would be frustrating for users not to be able to use the whole area of the new 13-inch monitors. So Apple implemented the "Command-key grow" feature that you

discovered, which allowed users to get whatever size they wanted. However, this feature was not without consequence to some applications, whose code couldn't handle larger-than-expected window sizes. The "Command-key grow" feature is documented on page 209 of the old *Inside Macintosh* Volume V, but wasn't documented in the newer *Inside Macintosh* series.

It's trivial to deal with this feature if you really need to limit the window size: simply check the size returned by GrowWindow, and if it's larger than you allow, reduce it to your maximum allowed size before calling SizeWindow.

Q *I've noticed that calling DeleteMenuItem mangles the menu data when handling menu items with strings that have between 251 and 255 characters. Does the Menu Manager have a problem when handling menu items with strings that long?*

A Yes, this is a problem, but it hasn't been documented in any *Inside Macintosh* books. The Menu Manager assumes that a menu item string isn't longer than 250 characters, so you shouldn't have menu items longer than that.

Q *When I use ShowDragHilite with a picture filling my window, it highlights only the areas that are the same as the background of my window. Is there any way to fix this?*

A Yes. ShowDragHilite isn't very savvy when overlaying image data other than the background color. The problem lies in QuickDraw's **hilite** mode. The operation of this mode is based rather coarsely on the background color. We're working on a fix for this problem, and eventually **hilite** mode will work significantly better in all cases, including that of selecting cells in lists drawn by the standard LDEF. Until then, your only alternative is to implement your own version of ShowDragHilite.

The question then becomes what color to use. Depending on your circumstances, you may want to use black, white, or perhaps even inversion, although you should try to avoid inversion against complex images if at all possible, since it can be ugly and confusing. Should you decide **hilite** mode is insufficient, it's up to you to decide how best to draw your highlight.

Q *Sometimes my application's calls to the Drag Manager fail with a -600 (procNotFound) error. This isn't one of the errors listed for these calls. What's up?*

A There are three known common causes of this error:

- The use of high-level debuggers — Since the Drag Manager interacts heavily with the Process Manager, as does the typical high-level debugger, conflicts inevitably develop. There's no workaround for this problem except to ask your debugger vendor to improve the debugger's behavior when debugging Drag Manager code. If your code is encountering such a problem, it should run fine when the debugger is not involved.

- Passing TrackDrag an event record whose **where** field is expressed in local coordinates — Such **where** fields often point outside the window in which the drag originates. (This can also cause a crash, but sometimes simply results in a -600 error.)

- Attempting to use the Drag Manager with Text Services Manager windows when the gestaltDragMgrFloatingWind bit isn't defined in the response to the gestaltDragMgrAttr Gestalt selector — The value of this bit denotes whether a Drag Manager bug with TSM windows is fixed on the system your application is running under.

In the last two cases, the Drag Manager has a hard time associating the source window with a process. Some operations can succeed even without a clear owning process, so the Drag Manager limps along as well as it can for a while in the hopes that it won't be asked to do anything that requires a ProcessSerialNumber. When it is asked, the operation fails.

Q *I've just implemented a DragDrawingProc. To start, I've tried simply to duplicate the default behavior of the Drag Manager (so that it will look as if I had not in fact attached a DragDrawingProc). Unfortunately, when the user drags into a valid drop area and the potential drop receiver calls ShowDragHilite, my DragDrawingProc seems to be responsible for leaving a trail of pixels on the screen. What am I doing wrong?*

A This happens because the Drag Manager doesn't always pass the entire "old" or "new" region to the DragDrawingProc. Below is a function that mimics what the Drag Manager does when you don't attach a DragDrawingProc to a DragReference before calling TrackDrag:

```
static pascal OSErr LikeDefaultDragDrawingProc(DragRegionMessage message,
    RgnHandle showRegion, Point showOrigin, RgnHandle hideRegion,
    Point hideOrigin, void *dragDrawingRefCon, DragReference theDragRef)
{
    OSErr       err = noErr;
    RgnHandle   xorMe;
    long        oldA5;
    Pattern     gray;

    switch (message) {
        case dragRegionBegin:
            oldA5 = SetA5((long) dragDrawingRefCon);
            gray = qd.gray;
            SetA5(oldA5);
            PenPat(&gray);
            PenMode(notPatXor);
            break;

        case dragRegionDraw:
            xorMe = NewRgn();
            if (!(err = MemError())) {
                XorRgn(showRegion, hideRegion, xorMe);
                PaintRgn(xorMe);
            }
            break;

        case dragRegionHide:
            PaintRgn(hideRegion);
            break;
    }
    return err;
}
```

The call to XorRgn is the key. It's also very important to pass the correct value for the dragDrawingRefCon to SetDragDrawingProc:

```
SetDragDrawingProc(dragRef, LikeDefaultDragDrawingProc,
                (void*)SetCurrentA5());
```

Note that the above works for 680x0 code; UniversalProcPtr creation has been omitted for simplicity.

By the way, be careful not to mix the use of SetDragDrawingProc and SetDragImage. See Technote 1043, "On Drag Manager Additions," for details.

Q *When my application calls the Drag Manager's TrackDrag routine and the user drags text out of my application onto the desktop, a clipping file appears. At least it does under System 7.5; under System 7.1, nothing happens. Why?*

A In Systems 7.1 through 7.1.2, the Drag Manager is implemented by means of multiple extensions (all in the Extensions folder), and various capabilities become available according to which extensions are installed. You can't count on any of these extensions being installed, so if you want your application to use the full functionality of the Drag Manager under these system versions, your application's installer should install these extensions.

Some systems may already have older versions of Drag Manager components, in which case you may want to replace them with newer versions. If you do, be sure to install all of the appropriate files to ensure version parity on the user's system.

Table 1 describes which components implement which functionality on which system. It's provided only for purposes of installation. Your application should not attempt to determine what functionality is available according to which files are installed (since users may have enabled some extensions without restarting, and since different versions of the system require different sets of extensions). Your application should instead test for Drag Manager functionality with the gestaltDragMgrAttr Gestalt selector.

In System 7 and 7.0.1, the Drag Manager is supported, but only for intra-application dragging. This makes it less desirable to install the required Macintosh Drag and Drop extension, because it provides nothing that can't be implemented through judicious use of QuickDraw, the Window Manager, and OSEventAvail.

In System 7.5, the picture is equally simple but significantly richer. All Drag Manager functionality is built into or installed with the system.

Table 1. Drag Manager files required for individual features

Feature	System 7.1	System 7 Pro (7.1.1)	System 7.1.2
Interapplication drag and drop	Macintosh Drag and Drop, Dragging Enabler	Macintosh Drag and Drop	Macintosh Drag and Drop
Drag and drop to/from Finder	Macintosh Drag and Drop, Dragging Enabler, Finder 7.1.3	Macintosh Drag and Drop, Dragging Enabler	Macintosh Drag and Drop, Dragging Enabler, Finder 7.1.3
Clippings	Macintosh Drag and Drop, Dragging Enabler, Finder 7.1.3, Clipping Extension	Macintosh Drag and Drop, Clipping Extension	Macintosh Drag and Drop, Dragging Enabler, Finder 7.1.3, Clipping Extension

Note: You should not install Finder 7.1.3 on user systems; in fact, there isn't an easily available license for shipping it. However, before System 7.5 it was the only way for developers without System 7 Pro (which includes Finder 7.1.3) to debug their code, so its use is documented here for historical reasons.

Q *Under MacTCP and Open Transport 1.0.x, if I'm using a Hosts file and I call AddrToName, the name resolves to the correct address. Under Open Transport 1.1 it returns an authNameErr. What's going on?*

A Open Transport version 1.0.8 mapped name-to-address and address-to-name translations into the same cache, and searched the cache whenever either a name-to-address or an address-to-name mapping was requested. Sounds good, right? The problem is, it broke several server load-sharing implementations that registered a service name as a single alias for a list of CNAMEs, each of which pointed to a server running the service. Under the former caching scheme, load-sharing techniques that depended on reverse lookups didn't work for the Macintosh — they'd always wind up with the same host name and hardware address for the original alias.

As a result, Open Transport 1.1 no longer caches address-to-name mappings (PTR records), nor does it search the name-to-address cache for address-to-name requests. (The treatment of CNAME records received was also modified, but that's irrelevant to your question.) Instead, it queries the configured domain name servers; apparently you got no authoritative information from any of them (or perhaps you weren't using them at all).

Strictly speaking, the behavior you're now seeing is more correct than what you saw before. DNS A resource records map names to addresses. To map an address to a name, you need a PTR record. The previous behavior of the MacTCP and Open Transport TCP/IP DNRs, treating the one as the mirror image of the other, was incorrect and has been changed accordingly.

The Macintosh Hosts file historically did not support PTR records, and does not support PTR records now. To do so, those records would have to be cached, once again breaking the load-sharing schemes. The only resource records the Hosts file supports are: A (name to address), CNAME (alias to fully qualified domain name), and NS (domain name server's fully qualified domain name). If you need a PTR mapping, you must register it with your local domain name server administrator, or maintain it within your own code from the results of your earlier name-to-address request.

Q *I'm writing an application using the native Open Transport APIs. My original intention was to ship only a 680x0 version of the application, but I've heard that this won't be compatible with future versions of Open Transport. Is this true?*

A Yes. We strongly recommend that you ship all native Open Transport applications as fat applications for maximum speed and compatibility.

Under System 7, Open Transport provides native APIs for both 680x0 and PowerPC clients. This allows 680x0 Open Transport clients to operate under emulation on a Power Macintosh. This won't be supported under future systems because they won't support the Apple Shared Library Manager, the dynamic linking technology used by 680x0 Open Transport clients. Table 2 summarizes this information.

Q *What are the different Gestalt selectors for MacTCP, Open Transport, and AppleTalk?*

A The Gestalt selector for MacTCP is 'mtcp'. MacTCP versions 1.0 through 1.0.3 didn't register this selector. Versions 1.1, 1.1.1., and 2.0 return 1, 2, and 3,

Table 2. Open Transport compatibility

Open Transport version	680x0 client on 680x0	680x0 client on PowerPC	PowerPC client
Open Transport 1.0.x	N/A[1]	Yes[2, 3, 4]	Yes
Open Transport 1.1, System 7	Yes	Yes	Yes
Open Transport 1.5, System 7	Yes	Yes	Yes
Open Transport, future systems	N/A[5]	No[6]	Yes

Notes:
1. Open Transport 1.0.x was never shipped or supported on any 680x0 Macintosh.
2. You must link with the Open Transport 1.1b6 or later libraries for this to work.
3. Obviously you must not call routines that were introduced with Open Transport 1.1.
4. Support for 680x0 clients on PowerPC isn't well tested under Open Transport 1.0.x. For this and many other reasons, you should implore your users to upgrade to 1.1.
5. Future systems will be PowerPC only.
6. Future systems won't support ASLM, so it's not possible for it to support Open Transport 680x0 clients.

respectively. If Open Transport is installed, 4 is returned. A value of 0 is returned if the driver is not opened.

The Gestalt selectors for Open Transport are gestaltOpenTpt and gestaltOpenTptVersions. You can test whether Open Transport and its various parts are available by calling the Gestalt function with the gestaltOpenTpt selector. The bits currently used are defined by constants in OpenTransport.h, as follows:

```
enum {
    gestaltOpenTpt = 'otan',
    gestaltOpenTptPresent = 0x00000001,
    gestaltOpenTptLoaded = 0x00000002,
    gestaltOpenTptAppleTalkPresent = 0x00000004,
    gestaltOpenTptAppleTalkLoaded = 0x00000008,
    gestaltOpenTptTCPPresent = 0x00000010,
    gestaltOpenTptTCPLoaded = 0x00000020,
    gestaltOpenTptNetwarePresent = 0x00000040,
    gestaltOpenTptNetwareLoaded = 0x00000080
};
```

If Gestalt returns no error and responds with a nonzero value, Open Transport is available. To find out whether Open Transport AppleTalk, TCP, or NetWare is present, you can examine the response parameter bits as shown above. For example, if you pass the gestaltOpenTpt selector to Gestalt, a result of 0x0000001F means that Open Transport is present and loaded, the AppleTalk protocol stack is also present and loaded, and the TCP protocol stack is present but *not* loaded.

The gestaltOpenTptVersions selector is used to determine the Open Transport version in NumVersion format. For example, passing the gestaltOpenTptVersions selector through a Gestalt call or MacsBug to Open Transport version 1.1.1b9 yields a result of 0x01116009. (Note that Open Transport versions 1.0 through 1.0.8 did not register this selector.) For more information on Apple's version-numbering scheme and the NumVersion format, see Technote OV 12, "Version Territory."

For AppleTalk, the Gestalt selectors are 'atkv' (no constant defined) and gestaltAppleTalkVersion. The gestaltAppleTalkVersion selector was introduced in AppleTalk version 54 to provide basic version information. Calling Gestalt with this selector provides the major revision version in the low-order byte of the function result. For example, passing the gestaltAppleTalkVersion selector in a Gestalt call or through MacsBug with a result of 0x0000003C means that AppleTalk version 60 is present. (Note that the gestaltAppleTalkVersion selector is not available when AppleTalk is turned off in the Chooser.)

The 'atkv' Gestalt selector was introduced as an alternative in AppleTalk version 56 to provide more complete version information via the 'vers' resource. For example, passing the 'atkv' selector to AppleTalk version 60 through a Gestalt call or MacsBug yields 0x3C108000.

Q *I'm writing an Open Transport module that conforms to the Transport Provider Interface (TPI). I find that Open Transport passes data to my TPI module using M_DATA message blocks, rather than M_PROTO message blocks with PRIM_type being T_DATA_REQ. What's going on?*

A The answer can be found at the end of the description of T_DATA_REQ in Appendix A-2 of *STREAMS Modules and Drivers* (UNIX Press, 1992):

> *The transport provider must also recognize a message of one or more M_DATA message blocks without the leading M_PROTO message block as a T_DATA_REQ primitive. This message type will be initiated from the write (BA_OS) operating system service routine.*

Open Transport deliberately uses this variant behavior as an optimization. By using M_DATA, it avoids allocating a buffer for the M_PROTO header. Since every memory allocation takes time, avoiding this one makes the system faster.

This behavior isn't seen on expedited data because the specification doesn't allow for this optimization on T_EXDATA_REQ.

Q *How can I launch a "foreground" task to run in the background?*

A You should use the LaunchApplication call in the Process Manager with the launchDontSwitch flag set in the launchControlFlags field. For more information about LaunchApplication, see *Inside Macintosh: Processes*, Chapter 2.

Q *Tackling a difficult problem in my application had put me in a foul mood, when a colleague pointed out that I was being awfully "tetchy." I told him the correct word is "touchy," and sure enough my spelling checker doesn't recognize "tetchy." Who's right?*

A Your colleague wins this one. The *Oxford English Dictionary* does recognize "tetchy," which means "easily irritated or made angry." It lists many variants for the spelling of this word, including techy, techie, teachy, teechy, tetchie, tecchy, titchie, tichy, titchy, tertchy, tatchy, and tachy.

AppendDITL Apoplexy

See if you can solve this puzzle in the form of a dialog between a pseudo KON (Bo3b Johnson) and BAL (Martin-Gilles Lavoie). The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And you'll also learn interesting Macintosh trivia.

BAL Well, KON, I'm very disappointed to announce that I've fallen back to the level of a newbie Mac programmer and am forced to ask you a question whose answer is probably obvious. However, I'm stuck.

KON An easy one? Great, we'll have it fixed in no time — and we'll give Puzzle Page readers a chance to get a decent score for a change. Details, please.

BAL I'm working on a new version of our application plug-in. One of the dialogs we had in the previous versions was very cluttered, and since we needed to add even more stuff to it, we decided to use "tabs" to group related options. But now the system crashes shortly after calling AppendDITL.

KON That's a well-traveled piece of the system. Why would you crash and no one else? Let's check for reported bugs…just as I thought, Developer Technical Support has no bugs listed for AppendDITL. Is the bug reproducible?

BAL 100% reproducible, and always after calling AppendDITL.

MARTIN-GILLES LAVOIE (mouser@zercom.net) constantly looks for more efficient means of using his hands. After experimenting with branch-prediction-enhanced, tristate-binary-enhanced, and floating-point-enhanced finger-coded binary techniques (pioneered by Tobias Engler in *develop* Issue 21), he went on to make a medieval ring mail consisting of over 26,000 rings. He hopes to make adequate use of his hands on his vacation in France, where he'll be touring medieval festivals and castles. Then he'll be back in Montréal-based Globimage, Inc., where he works on prepress-related utilities and automation software. •

BO3B JOHNSON (bo3b@apple.com), whose name is pronounced "Bob," has been programming the Macintosh seemingly forever and still hasn't lost interest (though it was touch-and-go there for a while). Having recently returned to Apple's Developer Technical Support group after a long stint as a dedicated slacker, windsurfer, and pursuer of arcane knowledge, Bo3b is rediscovering the joys of working for a living (and has actually found a few). Getting up in the morning, however, remains a serious challenge. •

KON When is this AppendDITL call made?

BAL The plug-in code runs as part of the host application and monitors activity within a floating window. One button in this floating window brings up a dialog containing a list of items that can be edited with a second dialog that comes up. The second dialog has tabs; when it's brought up, a call to AppendDITL is made to add the dialog items for the first tab panel into this dialog.

KON How does it crash, exactly?

100 BAL A little while after AppendDITL exits, the dialog is visually messed up, and the system drops into MacsBug with a bus error. Also, the application heap is usually corrupted.

KON This is a code resource, right? Code resources can't have A5 globals, so any globals would cause you to "color outside the lines." Are you using globals?

BAL The application plug-in interface file requires the use of globals, but it was built with CodeWarrior, which uses register A4 as a code resource's globals pointer.

KON What if AppendDITL trashes A4 during execution? That would cause some of your user item procedures to fail while trying to access globals.

90 BAL Well, let's check. When the dialog is first brought up and items are being appended to it for the default tab, it immediately crashes. Here, witness the disaster…

KON Indeed, it crashes really hard. MacsBug is alive, but the system is barely alive; I can't escape the application with **es**. Whoa! I can't even reboot with **rb**. Let's try that again, and watch AppendDITL. We do an **atb AppendDITL**, then trace over the call. Nope, A4 is untouched by AppendDITL. In addition, the heap was corrupted only after several AppendDITL calls, so I'd say A4 is solid and not the problem.

BAL Time for the Vulcan Nerve Pinch?

KON Control-Command-Power? Indeed! While we're rebooting, can I see the code where you add items to this dialog?

80 BAL My plug-in uses code that I've used in a standalone application. This code has worked just fine since then, and I don't see why it should make a difference when used in a code resource. It's pretty much by the book. When the user clicks the tabs in the dialog, we make a DITL content switch using this code:

```
// At this point, clickedTab contains a DITL ID corresponding to
// the clicked tab DITL ID. 'kPanelBase' is the number of items
// in the base DITL (ID 29000).
if (gCurrentPanel != clickedTab) {
   short numItems = CountDITL(theDialog);
   if (numItems > kPanelBase)
      ShortenDITL(theDialog, numItems - kPanelBase);
   Handle ditlResource = GetResource('DITL', clickedTab);
   AppendDITL(theDialog, ditlResource, overlayDITL);
   ReleaseResource(ditlResource);
   gCurrentPanel = clickedTab;
}
```

The first time around, the ShortenDITL routine isn't called because numItems is equal to (but not greater than) kPanelBase. I verified this at run time with source-level debuggers.

KON I'm wondering about the ReleaseResource right after the AppendDITL call.

70 BAL *Inside Macintosh* recommends doing this to avoid using altered DITL resources if we later display a dialog whose item list was previously altered. After the crash, I examined the content of the application heap to see if any expected resources were missing. I used **hd r** to dump all resource information in the current heap; I made sure that all my resources were loaded and that my code isn't inadvertently using the host application's resources.

KON Well, we know *Inside Macintosh* isn't always right, so let's display the DialogRecord.items handle after AppendDITL. We'll find this handle with **thePort** and then **dm . dialogrecord**.

BAL The size of the handle is different after the call, and it appears to make a copy of the DITL into this handle, so calling ReleaseResource shouldn't be a problem.

KON Did someone patch AppendDITL? Or perhaps the dialog underneath is causing a conflict?

65 BAL Let's look at AppendDITL with the debugger. Do an **il AppendDITL**. Hmm, the code is still in ROM at CommToolboxDispatch, so it's not patched. You also wouldn't expect any conflict between the two dialogs on the screen, because you have to pass the DialogPtr to AppendDITL, which gives it the exact DialogRecord.

KON OK, so far my ideas aren't panning out. But I notice that although it always crashes, it doesn't seem to crash the same way every time: sometimes it takes two AppendDITL calls and sometimes one.

BAL What could make it do that?

KON It's probably using some nearly random chunk of memory. Let's make it more reliable in the way it crashes by turning on heap scramble with **hs**. Do you want to use QC or Jasik's debugger instead? They're an even stronger way to make it more repeatable.

BAL No, this still seems like an easy bug, so let's stick with MacsBug. That **hs** seems to help.

KON As near as I can tell, there's some problem with the fonts. Most of the crashes happen while it's drawing text. It nearly always crashes in a TextBox or TESetText call, during the handling of DrawDialog. Do you change the fonts used to draw the dialog?

60 BAL I do. Here's the code:

```
((GrafPtr) theDialog)->txFont = Geneva;
((GrafPtr) theDialog)->txSize = 9;
if (*(((DialogRecord*) theDialog)->textH)) {
    // This will have to come from a resource for localization.
    (*(((DialogRecord*) theDialog)->textH))->txFont = Geneva;
    (*(((DialogRecord*) theDialog)->textH))->txSize = 9;
    (*(((DialogRecord*) theDialog)->textH))->lineHeight = 12;
    (*(((DialogRecord*) theDialog)->textH))->fontAscent = 10;
}
```

KON   Beats me. Intriguing question, though. Since it's only display code, let's comment it out.

55   BAL   It still crashes.

KON   Well, then it must not matter if we change the fonts in the window. By the way, BAL, while I don't think this is the solution to the bug, you should at least use the regular calls like TextFont and TextFace instead of setting the fields directly, and make TextEdit calls for the parts that modify the TEHandle.

BAL   Oh. Right.

KON   Time to look at a stack crawl? Let's try **sc**.

50   BAL   It says:

```
Start of link chain does not point to a stack frame
```

KON   Bummer. OK, how about doing an **sc7**, even though it's not as reliable? It won't miss anything, but it will show up a lot of junk addresses that aren't real. When you're desperate for information, you turn to **sc7**. Here, take a look… *(See Listing 1.)*

BAL   Geez, KON, could we make it any smaller for our *develop* readers?

KON   Hey, it's not critical for them to see the details. And our lawyers insisted on a full disclosure of all information related to finding the bug. Readers can always get out their magnifying glasses, if they want to.

BAL   Good thing it's high resolution or they'd have to call the psychic hotline instead of just squint. Hey, have you ever used one of those magnifiers on ants?

KON   BAL, can we focus on the problem at hand? Let's do the normal thing and skip all the unnamed routines, and just try to get a feel for what was going on. The routines that have a frame address in the second column are more likely to be valid calls; we can probably ignore the others.

45   BAL   Well, there's a call to MyWindowEvent+00294, and one to MyDoStyleList+00044, both my routines. Further down are my DoStyleListDialog+003F0 and MyDoStyleEditor+00080. Among the Toolbox calls of interest, there's one to _NewDialog+001C4, and later to _DrawDialog+0000A, and maybe in response to that a call to _TextWidth+00048.

KON   The TextWidth call makes me think the Font Manager might be damaged.

40   BAL   Further down, without any frame addresses, we get six KillPicture calls, two sIntRemove calls, two Jackson calls, and some other stuff that implies the computer ran off through the weeds for a while before it finally died. Hmm…there's no call to AppendDITL in the chain, though, and **sc7** wouldn't miss it.

KON   I thought you said it crashed in AppendDITL?

35   BAL   No; to be precise, I said it crashes shortly after AppendDITL, and if I comment out the AppendDITL call, the crash no longer happens, so it must be related.

Listing 1. The ant-sized **sc7**

```
Return addresses on the stack
Stack Addr  Frame Addr  ISA  Caller
0160A01E                68K  0160A420
0160A016    68K  01106010 'CODE 0001 1724'+00040
0160A208    68K  01106010 'CODE 0001 1724'+00040
0160A0F0    68K  40808028 AddDrive+00062
0160A0D0    68K  4084C20E SoundDispatch+0X370
0160A0F0    68K  0160A866
0160A7A0    68K  40800BEC _StripAddress+0047C
0160A7A4    68K  40800BF0 _StripAddress+00480
0160A7A0    68K  0003CF2A 'snd BFN1 010C'+0CCDEA
0160A794    68K  0003CF2A 'snd BFN1 010C'+0CCDEA
0160A7FC    68K  40818500 _SetRealLinkAttrs+00508
0160A8F8    68K  01117202 'CODE 006A 1724'+07052
0160A8D0    68K  0110A24A 'CODE 0075 1724'+0107A
0160A8B8    68K  01663656 MyWindowEvent+00294
0160A8A0    68K  01663624 MyDoStyleList+00044
0160A88C    68K  000C39A2
0160A888    68K  000C39A2
0160A878    68K  000C17BC
0160A874    68K  000C17BC
0160A868    68K  000C17BE
0160A860    68K  000C17BE
0160A0F4    68K  40819D58 _NewDialog+001C4
0160A0CA    68K  00B80030
0160A0BC    68K  40821824 _GlobalToLocal+0001C
0160A098    68K  0166TAF8 MODALDIALOGSUBFILTER+00046
0160A864    68K  000C291C
0160A056    68K  4081900A _NewDialog+00146
0160A052    68K  40819DCA _DrawDialog+0003C
0160A022    68K  40819DCA _DrawDialog+0003C
0160A002    68K  40800BEC _StripAddress+0047C
0160A07E    68K  40800BF0 _StripAddress+00480
0160A0CA    68K  4081587C _NewControl+0012C
0160A0A8    68K  40800BEC _StripAddress+0047C
0160A0A4    68K  40800BF0 _StripAddress+00480
0160A088    68K  00237E28 'CDEF 0000 0002'+00108
0160A084    68K  00237E28 'CDEF 0000 0002'+00108
0160A060    68K  0111b472
0160A05C    68K  40800BF2 _StripAddress+00002
0160A036    68K  40800BEC _StripAddress+0047C
0160A032    68K  40800BF0 _StripAddress+00480
0160A024    68K  40800200 _StripAddress+01598
0160A0CA    68K  000628A8
0160A0CA    68K  00C2032C
0160A0C8    68K  000C032C
0160A002    68K  00C2093C
0160A07E  0160A67A  68K  0168AC24 DoStyleListDialog+003F0
0160A0E8  0160A664  68K  01663902 MyDoStyleEditor+00080
0160A060  0160A03C  68K  00074536
0160A024    68K  000C39A2
0160A050  0160A6AC  68K  000C39A2
0160A040    68K  000C17BC
0160A03C  0160A438  68K  000C17BC
0160A028    68K  000C17BE
0160A0AA    68K  000C17BE
0160A09C  0160A988  68K  40819D58 _NewDialog+001C4
0160A0A2  0160A996  68K  01663C98

CORE INTERFACES STYLEEDITORFILTER+00022
0160A994    68K  0160A0C8
0160A09E  0160A95A  68K  0160A974
0160A996  0160A952  68K  0015673A
0160A091    68K  40819CC9 _NewDialog+00134
0160A912  0160A90E  68K  4081A634 _FindDialogItem+0630E
0160A902    68K  000C25A0
0160A0FE    68K  000C25A0
0160A902    68K  40831440 _Jackson+00010
0160A98E    68K  40836188 _SetRectRgn+00018
0160A874    68K  40819D9C _DrawDialog+0003E
0160A860    68K  40819D98 _DrawDialog+0000A
0160A90A  0160A0036  68K  4081A13C _FindDialogItem+00239
0160A7D8  0160A7D4  68K  40825DFC _GetNewBar+00270
0160A7A8  0160A97A4  68K  40829228 _TextWidth+00048
0160A7A  0160A706  68K  4082C168 _StdDevLong+0012E
0160A77C  0160A7778  68K  4082C10C _rRlite+0033E
0160A0B2    68K  0506A16
0160A0A8    68K  0506J2A2
0160A91C    68K  0019E172
0160A500    68K  40819D92 _v84dExit+001C8
0160A5B0    68K  40800BEC _StripAddress+0047C
0160A5AC    68K  40800BF8 _StripAddress+00488
0160A5A2    68K  00312744
0160A536    68K  40833ADA _KillPicture+0004A
0160A528    68K  40832B88 _KillPicture+002F8
0160A526    68K  40832B9C _KillPicture+002C
0160A520    68K  40832B9C _KillPicture+002C
0160A50A    68K  40832B9C _KillPicture+002C
0160A50A    68K  40833D36 _KillPicture+000A4
0160A04C    68K  40802008 _StripAddress+01558
0160A0C8  0160A04C4  68K  00285E84
0160A04E    68K  40833D9C _KillPicture+002C
0160A04AC    68K  40803B68 _UningTrap+00008
0160A04A1    68K  40804ED4 _SIntRemove+0005C
0160A04C    68K  40804FD4 _SIntRemove+0008A
0160A0488    68K  4080A348 _DoVBLTask+00016
0160A458    68K  40800FFF _v0txn414+0008A
0160A04C    68K  40800FFF _v0txn9414+0008A
0160A03C0    68K  8003C380
0160A03DC    68K  40831450 _Jackson+00010
0160A093DC    68K  40831450 _Jackson+00010
0160A03170    68K  40800C9E _StripAddress+0051B
```

KON	OK, but I still think the Font Manager is trashed, because when I look in low memory at CurFMSize, I see the font size is 4583! That's a bit big for this computer.
30 BAL	Not only do fonts get smashed, but sometimes the dialog's added items will draw with wacky colors. It appears that the whole graphics port (the current window) gets written over sometime during the AppendDITL call.
KON	Hey, there's Dave Polaschek and Quinn. Dave's run into problems with AppendDITL before, so let's ask him about it. Dave, what exactly does AppendDITL do?
Dave	Nothing too tricky: It locks the new DITL handle and then steps through the items in the list, loading and installing each one into the dialog. It uses GetNewControl for controls, copies in the static text items, and loads and puts the handles to PICT resources in the DITL. When it's done, the new DITL handle is unlocked but left in memory. There's an old version of the source code in Technote PR 09, "Print Dialogs: Adding Items."
KON	Any nasty behavior or bugs that you know about?
Dave	The only common problem I've heard of is with 'ictb' resources. Are you using them to specify fonts and sizes or color tables for your DITLs?
25 BAL	No. Also, no font-specific action is taken at run time — I don't call the Font Manager, and I don't directly call ATM. The only thing related to fonts that I can see is that the dialog's font and size are changed to Geneva 9 before the AppendDITL (which goes against guidelines for localization). Wait, there's also a font pop-up menu that's being updated with the current font list when the dialog gets called up. But both these things worked before I started playing with AppendDITL.
KON	Still, those 'ictb' resources sound suspiciously like what we're seeing. Quinn, have you heard of any bugs relating to 'ictb' resources?
Quinn	Nothing specific, just what Dave said: they're more trouble than they're worth. Have you looked for 'ictb' resources in the heap? Try doing an **hd ictb**.
KON	Hey, there are a bunch in there! How about in the resource fork? I'm doing an **rd ictb** too.
20 Quinn	Sure enough, there are some in the resource fork. Those 'ictb' resources are only used to change the look of the dialogs; they aren't required. Since we don't trust those things, why don't you delete them in a copy of the plug-in? Where's ResEdit?
KON	OK, I cleared them out; now let's try it again. Dang! It still crashes, and the same way. I would have bet that was it. How many points do I have left?
Quinn	Not enough to clear your karma point deficit. Why don't you do an **atb DrawDialog**, and then just an **atb** so that we can see what happens before it dies. We won't see any PowerPC calls, but this is all 68K code anyway.
KON	OK. The last dialog-related thing it did was call TETextBox to draw one of the static text items — another hint that it's a font problem.
Dave	To find this, you'll need to reduce the number of variables. See if having fewer items in your dialog changes anything. Maybe a corrupted item

or a resource conflict is causing the problem. You said you're using at least one pop-up menu, right? Is its menuHandle properly loaded?

15 BAL I tried a number of variations on what you've suggested; I simplified the code in all these ways:

- I disabled any filter procedures passed to ModalDialog.

- I removed any user item procedures.

- I removed any code-handling items in tab panels.

- I converted (one by one) all items in the first tab panel to static text items, relaunching the host application every time.

- I reduced the number of items in the first tab panel, down to six static text items.

- I removed any code having to do with the dialog's base elements (those that stay regardless of the current tab).

- I converted the base dialog's elements to static text.

- I moved items around so that appended items wouldn't be surrounded by PICTs that form the frame of the tab area.

None of this worked. In all cases, the same bad behavior occurs after I come out of AppendDITL. I even tried using just a single plain button and one static text item, but it still trashed the heap. Furthermore, all the pop-up menus in this dialog (and its tab panels) use different menus, and all menus are loaded and installed during startup, with InstallMenu(theMenuHandle, -1).

KON Well, this is getting even more interesting now. I'm willing to bet that this is a bug in AppendDITL, but I can't put my finger on it just yet. To simplify even further, let's make a small test application with only the code that handles the dialog.

Dave Good idea. But I'm hungry, so I'll leave you to it. Anybody want to have dinner?

Quinn I'll join you — though I like the taste of freshly killed bug better.

KON That leaves it to us, BAL. We're making strong progress now, though, so we should have this one crushed in an hour or two.

BAL KON, we've been looking at this for over five hours now, we're nearly down to 10 points, and we still can't find it.

KON I'm just going to make a simple application that creates the dialog using your code. And I'll copy the resource fork of your plug-in so that we get all the DITLs. I want to rule out the host application having any effect.

10 BAL OK, excellent! And it still crashes with the test app. Now we can keep simplifying until we find the offending code.

KON Let's take a big simplifying jump and change the dialog to just a control and a static text in each tab panel, as in your test.

BAL Huh! It still crashes in this nearly trivial case. I guess we can't blame the host application.

KON Now let's keep trying to find where it stops crashing. I've removed the entire content of the tab panels' DITLs and replaced their content with a single plain button straight out of ResEdit's tool palette.

Each button on each tab panel says something different, just to make sure they're actually removed when they're tabbed out.

BAL   Look at that, it worked! I doubt we have a limit of one item per AppendDITL — that would be ridiculous.

KON   I agree. So it must have something to do with static text. Fonts, I tell you! Fonts, fonts, fonts, fonts, fonts!

BAL   Easy, KON. We'll find it. Let's put the static text items back to make it crash again.

KON   OK, now let's trim the icons and other junk out of the resource fork, to be sure it's not interfering, and to make it as simple as we can get yet still crash.

5   BAL   Hey, there are 'ictb' resources in our test app!

KON   They must have come from the plug-in when we copied the resource fork.

BAL   And look at this. My plug-in file has grown in size between its installation and its first run in the host application. I don't modify my own code in this plug-in — something's fishy.

KON   You said you weren't using any 'ictb' resources, yet earlier we saw lots of them in your plug-in.

BAL   Just a second, while I call the host application's manufacturer… Ahem. Get this: They warn me to make sure to have an 'ictb' resource for every 'DITL' resource in a plug-in file. If any are missing, the host application adds empty 'ictb' resources for what it thinks are "deficient" plug-in files. I'm very curious about why the host application requires the presence of 'ictb' resources in plug-ins.

KON   D'oh! The host application is modifying your resource fork? That explains why our 'ictb'-deleting experiment didn't work. Sure enough, if I delete the 'ictb' resources from the test app, it works fine.

BAL   Obviously the host application doesn't know I'll be using some of those DITLs to expand another DITL. The Dialog Manager ends up having too few 'ictb' entries for the number of items in the expanded dialog, and when it gets to the end of the 'ictb' resource that the host application created, it starts reading garbage from memory, trying to set fonts, sizes, and colors.

KON   The Dialog Manager sets up the 'dctb' and 'ictb' data structures only when the dialog is created, and doesn't change them for AppendDITL? That does it, I'm filing a bug against AppendDITL. Let's see, that's #1377732.

BAL   The static text items are a bit more fragile in this case, which is what we found with that test of having one static text item. Buttons just have a ControlRecord — and a color table, which only causes funny colors to appear if the Dialog Manager is using random bytes out of memory.

KON   Of course! For static or editable text items, it blindly goes off the end of the handle, using whatever junk is in memory as a txFont, txFace, txSize, or color table. When it set the font to number 43003, the size to 15433, and so on, the Font Manager was none too pleased.

BAL   I added an empty 400-byte 'ictb' resource to my project, with the ID of the base dialog. This is enough to accommodate this dialog's items, plus any items I add through AppendDITL. Everything works fine now.

KON Hot dog! I knew we were going to crush this thing before too long. You have no idea how glad I am that this one has been killed. Well, actually, I bet you do; it stumped us for seven pages.

BAL The host application's manufacturer also told me they need to add 'ictb' resources in order to solve an old problem where plug-ins used dialog windows with the same IDs as some of their application's dialog windows. When these dialog windows were loaded by the system, the system would look for its associated 'ictb' resource with GetResource, which looks through the resource chain until it finds one. Sometimes it would pick an 'ictb' in their application, which wasn't suitable for the plug-in's dialog window, causing the same kind of problems we've experienced with my plug-in. Watch me never assume what's in my resource fork from now on!

KON Nasty.

BAL Yeah.

INDEX

RESOURCES

Apple provides a wealth of information, products, and services to assist developers. The Apple Developer Catalog and Apple Developer University are open to anyone who wants access to development tools and instruction. Additional information and services are available through Apple's Developer Programs.

The Apple Developer Catalog offers worldwide access to Apple's development tools, resources, training products, and information for anyone interested in developing applications on Apple platforms. This complimentary catalog features Apple development products and offers convenient payment and shipping options, including site licensing. The catalog also includes a directory of third-party products.

Apple Developer University (DU) provides courses to get you started programming on Apple platforms, as well as advanced, in-depth training on new technologies such as QuickTime VR, OpenDoc, and Apple Media Tool.

In addition to classroom training, self-paced courses are available through the *Apple Developer Catalog*, and free introductory tutorials are provided on the DU Web site, at http://www.devworld.apple.com/dev/du.shtml.

The Macintosh Developer Program provides members with ongoing Macintosh-related technical information and services. It includes:

- The Apple Developer Mailing, which includes the *Developer CD Series*.
- Macintosh technology seeding.
- Programming-level technical support via e-mail. Apple offers a number of options for varying levels of technical support.

The Newton Developer Program provides ongoing Newton-related technical information and services. It includes:

- The monthly Newton Developer Mailing.
- The quarterly Newton Developer CD.
- Newton development class discounts.
- Programming-level technical support via e-mail. Apple offers a number of options for varying levels of technical support.

The Apple Media Program (AMP) provides resources to keep multimedia developers up-to-date on Apple's offerings for authoring and playback. For more about the benefits and resources of this program, visit the AMP Web site at http://www.amp.apple.com.

Apple Developer Catalog To order a product or receive a catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. You can also send e-mail to order.adc@apple.com, or write *Apple Developer Catalog*, P.O. Box 319, Buffalo, NY 14207-0319. The *Apple Developer Catalog* is also on the Web at http://www.devcatalog.apple.com.

Apple Developer University Course descriptions and schedules can be found at http://www.devworld.apple.com/dev/du.shtml on the Web. You can also call (408)974-4897, fax (408)974-0544, send e-mail to devuniv@apple.com, or write Developer University, Apple Computer, Inc., 1 Infinite Loop, M/S 305-1TU, Cupertino, CA 95014.

Apple Developer Programs These programs vary on a country-by-country basis. For more information on any of Apple's developer support programs worldwide, call (408)974-4897, fax (408)974-7683, or send e-mail to devsupport@apple.com. Also see the Developer Programs Web site at http://www.devworld.apple.com/dev/programs.shtml.